POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Automated testing plan for encrypted communication protocol

Definition, development and implementation of an automated testing plan to validate the CAN-FD architecture (flexible data rate) and the message encryption system MAC/ARC (Message Authentication Code / Anti-Replay Counter) of an ECU

Supervisors
Prof. Massimo Violante

Candidates
Fiorenza Conte
251535

Internship Tutor
dott. ing. Antoine Poitrenaud

Anno accademico 2019-2020

A mia madre e a mio padre.
Comunque vada, sarà un successo.

Abstract

In the automotive industry, one can observe an impressing increase of electronics presence which is reflected in the presence of Electronic Control Units (ECU) and the communication protocols that connect these.
Furthermore, the introduction of large numbers of sensors to provide driver assistance applications and the associated high-bandwidth requirements of these sensors have accelerated the demand for faster and more flexible network communication technologies within the vehicle.
One of the latest technologies to overcome this issue is CAN-FD, a new release of the well-known communication protocol CAN. The latter was invented and commercialized by Bosch GmbH in 1982, while the former was released in 2012.
This work was conducted in collaboration with the french branch of Bosch GmbH, in the site of Saint-Ouen. The goal of this project is to develop automated test plan for the CAN-FD. Following the Open Systems Interconnection standard, we can say that this work covered the testing of the data link layer of this communication protocol for the OEM Renault Nissan.
In the first phase of the project, the testing plan was designed and developed starting from the system requirements of the OEM. These requirements were to be applied to the CAN-FD frames (i.e. the primary message format). The requirements were divided into TX requirements, those for the frames sent by the ECU under test, and RX requirements for the frames sent in the other direction.
Since automated testing was already developed and applied to CAN, it was convenient to re-use the existing testing plans, after having properly adapted, to the new protocol.
For this purpose, it is worth mentioning the most remarkable differencies between CAN and CAN-FD:

1. the possibility of changing the data-rate of transmission and the consequent improvement of throughput;

2. the different length and structure of the data field, i.e. the part of the frame where the message is transmitted without considering all the synchonization and control part,

3. the presence of a encrypted authentication system for certain frames called MAC/ARC.

Point 2 is the most important aspect to deal with when adapting the already existing CAN testing plan to the CAN-FD.
The testing plans were realized and run on the tool ECU-TEST by TraceTronic. Here, for each of the testing requirement, a test package was written. The student was in charge of adapting and developing new test packages, by writing python scripts to be inserted during the stimulation and/or analysis phase.
During the second phase of this project, the job of the student was to make the generation of the whole set of frames automated.
This was done by developing a workflow on the internal tool TestFlow. The workflow was supposed to:

- take and analyse several description files,
- extract the frames to test,
- divide those into groups for specific properties,
- generate the complete testing plan on ECU-TEST, including configuration files and links.

For this task, the student had to write, for each of the step above, several scripts in QVTO language, a model transformation language, conform to Meta-Object Facility (MOF) 2.0 metamodels.
The third phase of the project was to build the test plan for the validation of the authentication system MAC/ARC. Here, the student developed another ECU-TEST package for it, with another python script aimed to compute the expected authentication code, following the AES-CMAC algorithm, and comparing the result with the received one.
The conclusion of this project was to find at the end of the second phase come bugs which were promptly corrected.

Contents

List of Figures

Chapter 1

Introduction

In the automotive industry, one can observe an impressing increase of electronics presence which is reflected in the presence of Electronic Control Units (ECU) and the communication protocols that connect these.
Furthermore, the introduction of large numbers of sensors to provide driver assistance applications and the associated high-bandwidth requirements of these sensors have accelerated the demand for faster and more flexible network communication technologies within the vehicle.
In the following chapter, a brief overview of the main aspects related to Electronic Control Unit and the networks that connect those are proposed.

1.1 Automotive Electronic Control Unit

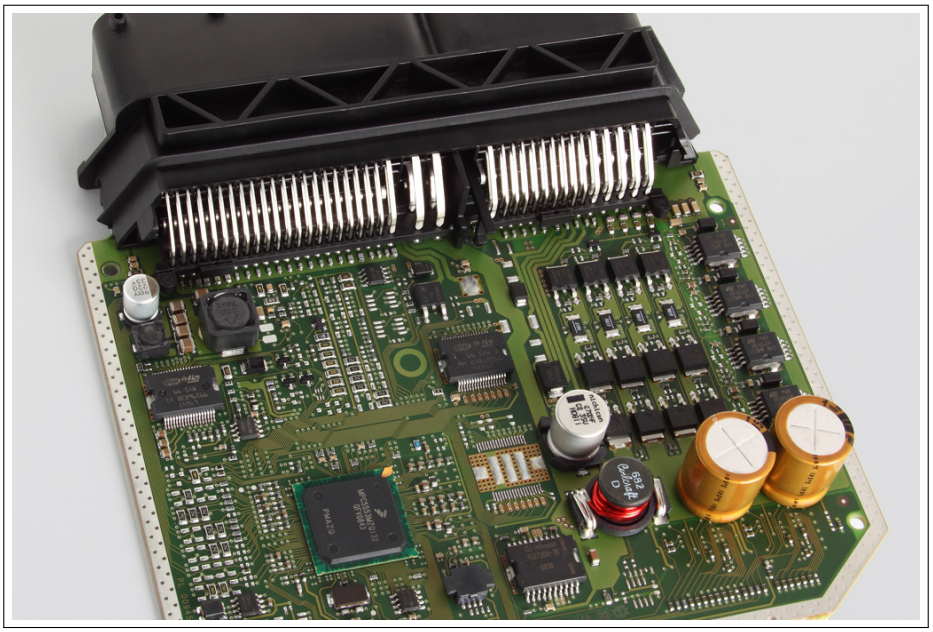

Figure 1.1. ECU [7]

Automotive industry is using more electronics to manage with the diversifying requirements of drivers and passengers and to tackle with concerns about the environment and fuel consumption. In a nowaday car we can find out a plenty number of Electronic Control Units, from 20 to 100. It is called ECU "any embedded system in automotive electronics that controls one or more of the electrical systems or subsystems in a vehicle". [8]
Their increasing number is leading to create of a sort of distributed intelligence within the car. Therefore, all the ECUs are interconnected by means of networks and cooperating to provide safety and efficient driving experience.

Inside a vehicle there are different types of ECUs; the main differences are related to the system to control: [12]

- *Engine control module*: Also known as an engine control unit. Responsible for assessing the load of the engine and tuning the ignition, fuel delivery and more to deliver optimum performance and economy.

- *Transmission control module*: These control the way, and when, an automatic gearbox shifts. Besides being fed with sensor data from the transmission itself, TCMs may also take data from the engine control unit to deliver more suitable, precise shifts.

- *Suspension control module*: Sometimes dubbed a ride control module and common in active,

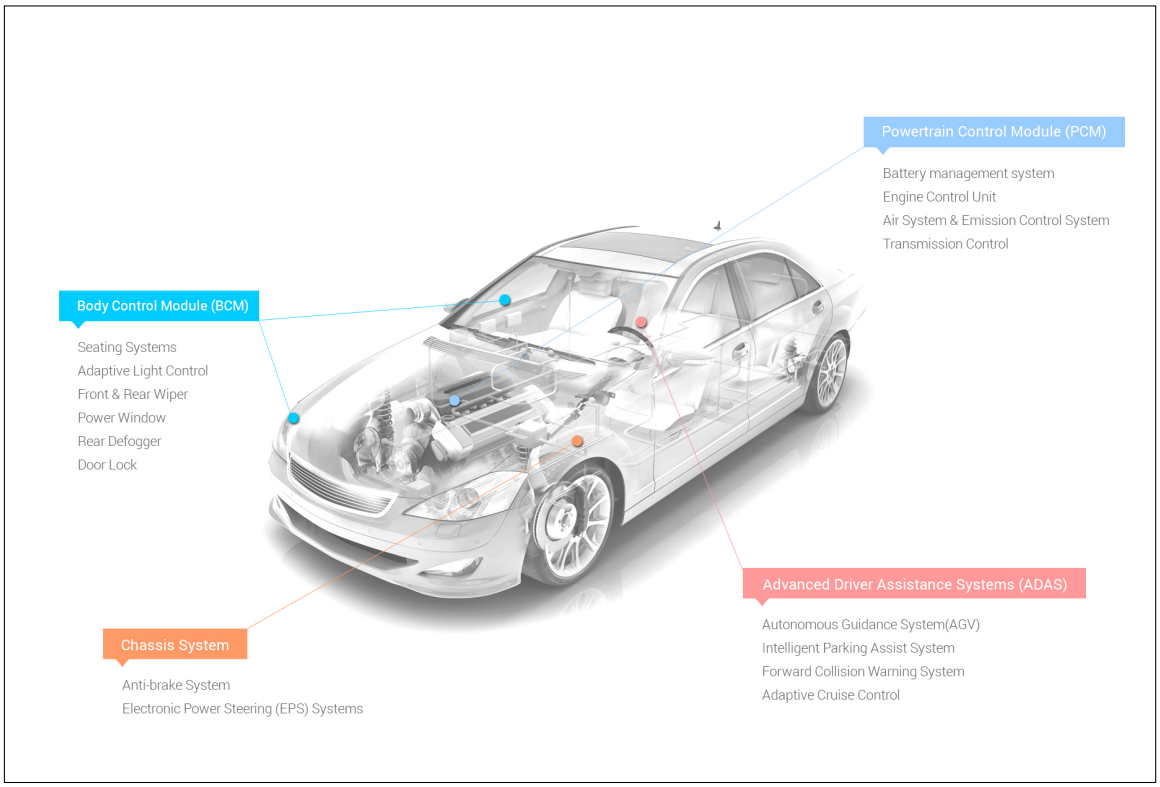

Figure 1.2. Automotive ECU in a car [2]

adjustable or air suspension set-ups. These adjust the suspension to suit the current driving conditions, or work to maintain the correct ride height.

- *Body control module*: This unit is typically responsible for controlling the car's myriad electrical access, comforts and security features. Common features it controls include door locks, electric windows and climate systems.

- *Telematics control module*: Typically offers internet and phone connectivity for the car's on-board services. May also include a GPS receiver for navigation services.

The different nature of the systems to be controlled prescribes specific requirements for each ECU; hence, different kind of networks. Also, since ECUs are mainly real-time systems, data must be provided within a certain deadline: communication must be deterministic (on time) and reliable (safely).

1.2 V-cycle

In automotive industry, due to market laws, ECUs software has to be developed and tested in a fast and fault-free manner. The most popular design and testing working flow is the V-shape model or V-Model.
It is compliant with the **ISO 26262**, which applies to safety-related road vehicle electronic and electrical (E/E) systems, and addresses hazards due to malfunctions.

The V-Model "demonstrates the relationships between each phase of the development life cycle

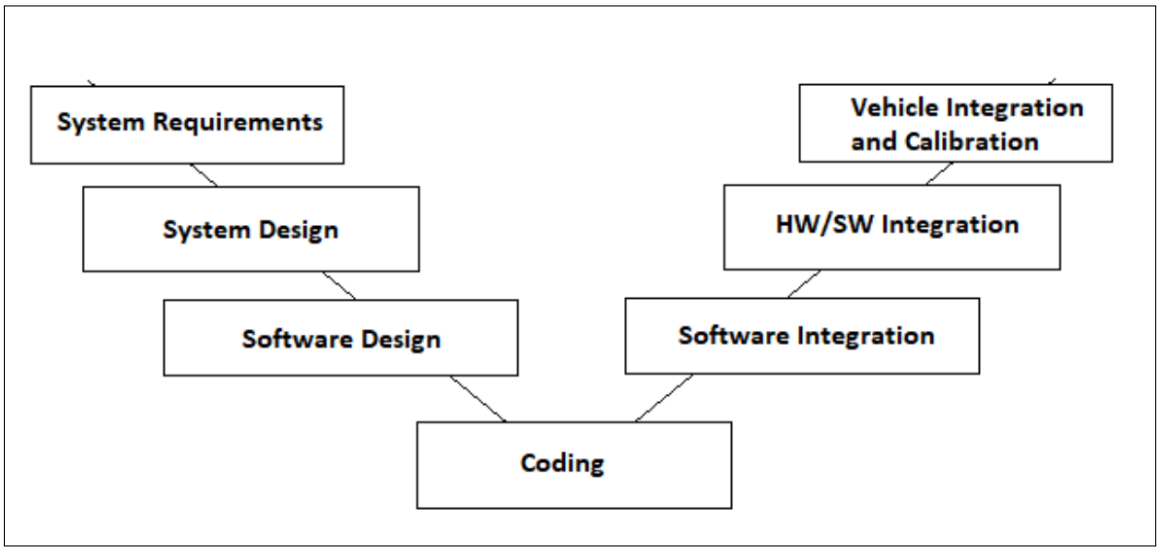

Figure 1.3. V-shape model development flow

and its associated phase of testing" [11]. Referring to figure 1.3, we can see that this is done by "folding" the waterfall life cycle into two branches, the left one for the design phase and the right one for the validation phase.
The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.
To each of the phase on the left branch there is the corresponding validation phase:

1. **System Requirements**: in this phase, item definition is defined. Hence, the functionalities to be provided in order to achieve functional safety.
2. **System Design**: partitioning of the functionalities into submodules that shall be designed and then implemented.
3. **Software Design**: definition of the functions to be used for implementing the subsystems functionalities.
4. **Coding**: implementation of the instructions of the previously defined functions.

5. **Software Integration**: merging of the different coded subsystems.

6. **HW/SW Integration**: integration of the whole software in the embedded hardware (execution platform) and testing the behavior.

7. **Vehicle Integration and Calibration**: joining the item into the vehicle and validating its functionalities. Furthermore, if the item is meant for being used by different vehicles, re-calibration is performed taking into account specific parameters.

Phases 1, 2 and 3 belong to the left descending branch, which describes the design process. Instead, on the right ascending branch, we find the validation process, described by the phases 5, 6 and 7. As already mentionned, we can see an orizontal correspondence between the design and validation phases: the System Requirements (1) design phase is validated by the Vehicle Integration and Calibration phase and so on and so forth. Hence, whether an error is discovered in a certain phase of the ascending branch (i.e. 6), it is necessary to go back to the corresponding phase of the descending branch (i.e. 2). The later the error is detected, the more expensive the designing phase is.
In order to avoid unintended coding and requirement faults, **automatic code generation** and **model-based development** are used.
The automatic code generation is very efficient in terms of productivity, even if the obtained code needs to be optimized. Once the model has been tested, it is sent to tools (i.e. Simulink) for automatically code translation.

1.2.1 X-in-the-loop

The requirement faults avoidance is performed by considering the **X-in-the-Loop** approach. This methodology allows to meet the requirements of the item by performing early phase development check simulations, hence virtually. This approach can significantly reduce the costs of both design and validation phases, as well as anticipating the detection of wrong implementations.
The 4 main simulation methodologies are:

Model-in-the-loop

Its purpose is to reach the most accurate implementation of the controller. The latter is the subsystem to be translated from model into code. Referring to figure 1.20, this step is applied during the phases 1, 2 and 3.
Here, the model of both the controller and the plant exists entirely in native simulation tool (i.e. Simulink or Stateflow). The goal is that "only the controller with unknown dynamics need be physically tested, reducing the cost and complexity of the physical test apparatus" [**?**]. This way,

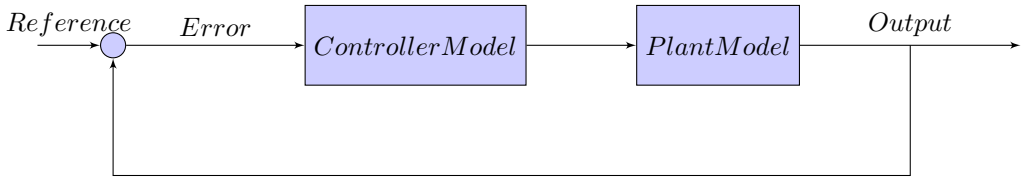

Figure 1.4. Model-in-the-loop scheme

the test speciment can be tested even if some parts have yet to be physically realized and the entire feedback control system is run in a **platform-independent way**, i.e. running both models on the same development PC. A scheme is shown in figure 1.4.
A reference signal is given as input to the feedback chain and the error signal is computed. The latter is set as input of the controller, which gives the control signal. Hence, the plant is fed with the controller output and the response to control signal is evaluated.
The behaviour must be compliant with the system requirements. If it is not, then the controller must be designed from the beginning. Otherwise, the automatically generated code of the controller can be tested, by the next step called "Software-in-the-loop".

Software-in-the-loop

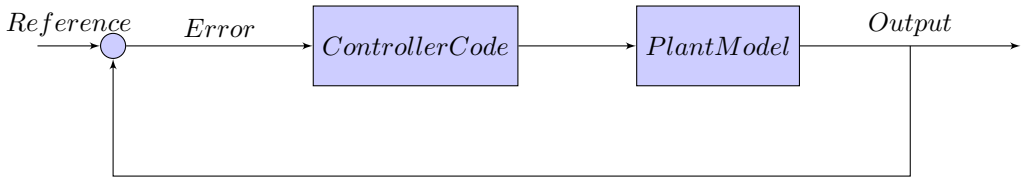

Figure 1.5. SW-in-the-loop scheme

Once the control model is tested, the control code can be automatically generated and optimized.
In this phase, details related to the hardware platform can be added (i.e. on which type of processor the code is deployed). Indeed, if during Model-in-the-loop a platform-independent approach was used, in SW-in-the-loop, the controller software is forwarding to run over a **specific platform**.
This implies that the software must be *adapted for that target hardware* making the code **platform-dependent** but more implementation friendly.
In the SiL process, the controller is translated into code and it is co-simulated with the plant model in order to check if the optimized code is compliant with the model behavior. It is important to highlight that the simulated modules run over the same environment (laptop).

Process-in-the-loop

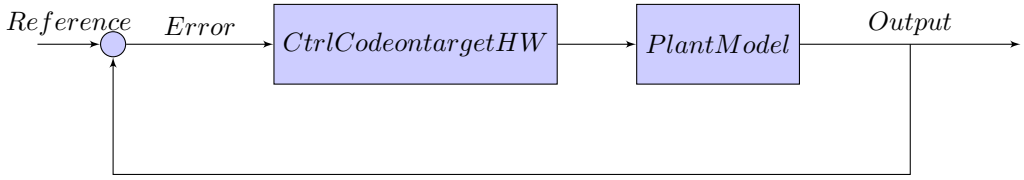

Figure 1.6. Process-in-the-loop scheme

In this part of the process, the controller code runs over a real embedded hardware, which will be used for the application.
Referring to figure 1.6, the plant is still simulated by a model on the development platform (i.e. Simulink environment), but it interacts with the rapid prototyping hardware, or ECU, or an Evaluation Board (EVB).
Therefore, this phase is very close to the real implementation. As a matter of fact, it aims to validate whether the platform-dependent code works properly in compliance with hardware requirements. The board pins take incoming signals and discretize them by Analog-to-Digital converters and Sample and Hold. Hence, they are translated to digital quantities and processed, in order to compute an output to be sent to the simulated plant.
Nevertheless, due to simulated plant, the time spent for running that simulation can be faster or lower than actual system. Thus, the control system is still not working in real-time: in order to reach real-time validation, the ECU needs to be mounted over the vehicle.

Hardware-in-the-loop

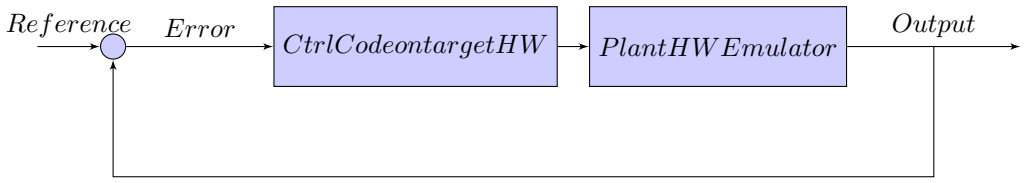

Figure 1.7. HW-in-the-loop scheme

This is the last phase of the validation process. Referring to figure 1.7, the control algorithm runs over the ECU, while the plant is emulated by an **emulation hardware**. It is worth remembering that an emulation hardware is "hardware that enables one computer system (called the host) to behave like another computer system (called the guest)" [**?**].
The emulation hardware is used in this step as it is able to run in real-time. Moreover, as the controller is not able to distinguish from the real and the emulated plant, the harness is the same

expected in the final application. It is necessary to highlight that 1 second in the emulator equals to 1 second in the real plant.
The HiL phase needs a lot of effort by test engineer, since it is one of the last step before launching the item on the market. Once the HiL testing phase is performed, the item is ready to be mounted over the vehicle for the calibration and integration.

1.3 AUTOSAR

AUTomotive Open System ARchitecture AUTOSAR is standardization initiative of leading automotive manufacturers and suppliers. Its goal is the development of a *reference model architecture for ECUs software* that can manage the **growing complexity of ECUs** in modern vehicles.
The main benefits of AUTOSAR are:

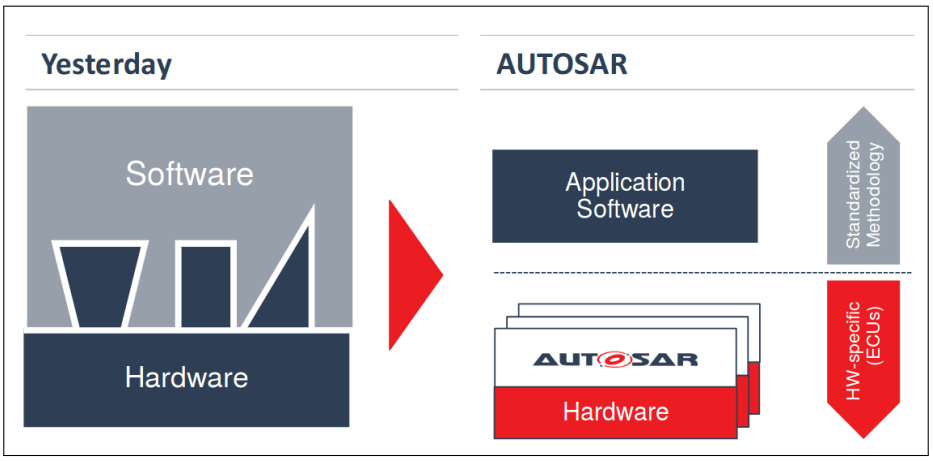

Figure 1.8. AUTOSAR layers

- Hardware and software widely independent of each other, as shown in figure 1.8.
- Development can be de-coupled by horizontal layers, reducing development time and costs.
- Decoupling the layers, it is easier to enhance the reusability of the software, which improves quality and efficiency.

The main working topics of AUTOSAR are:

- The **software architectures** including a complete basic software stack for ECUs, *AUTOSAR Basic oftware*, as an integration platform for hardware independent software applications.

- The **methodology** that defines the exchange formats and description templates to enable a correct and coherent configuration process between the basic software stack and the integration of application software in ECUs.
- The specification of **interface** of typical automotive applications (i.e. CAN), as a standard for application software.
- The specification of **test cases** intending to validate the behavior of an AUTOSAR implementation with AUTOSAR application software components or within one vehicle network.

1.3.1 AUTOSAR Architecture

The AUTOSAR Architecture, shown in figure 1.9, distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller. Also, it defines the interfaces between these components, such that the

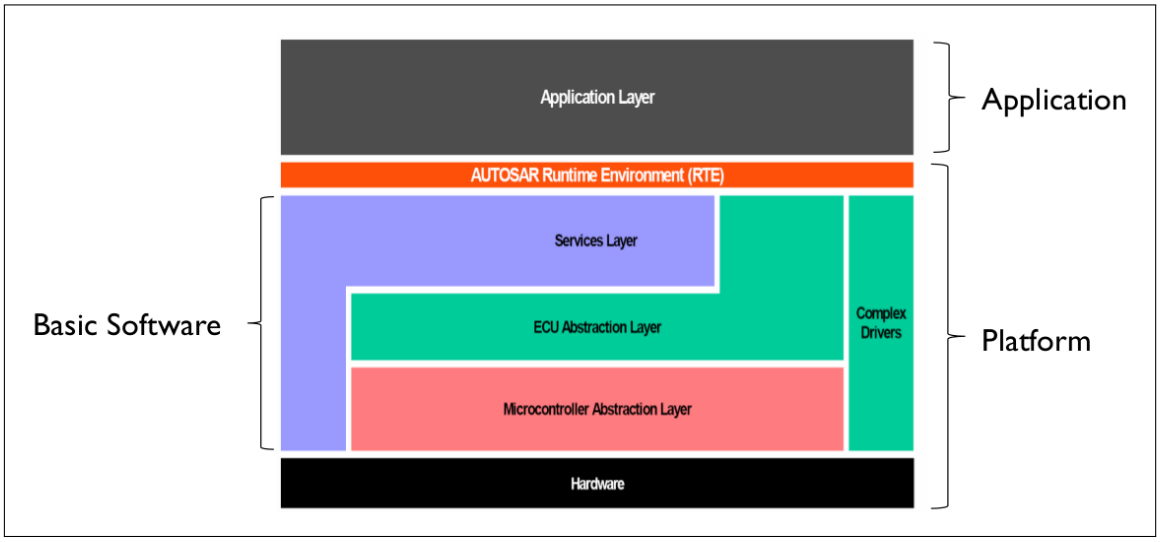

Figure 1.9. AUTOSAR classic architecture

development of each component and their integration can be demanded to different suppliers. Going deeper in the definitions of the three software layers:

- The **Application** is where the AR application is defined. The application is defined as set of software components (SwCs) that communicate with other components and/or services via the RTE. Each SwC is defined through:
  - port interface defining how the SwC communicate with other SwC and the system services,
  - internal behavior defines as runnable entities(e.g., tasks).

- The **RunTime Environment (RTE)** provides communication services for the application software. Its task is to make the SwCs independent from the mapping to specific ECUs. Moreover, the RTE is generated automatically from the SwCs mapping on the ECUs
- The **Basic Software(BSW)** provides standardized software modules that offer services necessary to run the functional part of the upper software layer. Its description file is the .arxml, which will have an important role in this work (2.3.1).

1.4 Automotive Networks

Modern vehicles generally employ a number of different networking protocols to integrate a growing number of ECUs into the vehicle. The introduction of large numbers of sensors to provide driver assistance applications and the associated high-bandwidth requirements of these sensors have accelerated the demand for faster and more flexible network communication technologies within the vehicle.
Due to the absolute need for reliability and security in safety-critical systems, wired solutions are expected to dominate for the foreseeable future. To overcome the problem of increasing complexity in point-to-point connections between ECUs, multiple ECUs are connected to one another using bus-based networks such as controller area network (CAN) or FlexRay.
With the increasing number of nodes connected to a bus, the bandwidth consumed significantly increases as well. As said in [22], the question of bandwidth started to manifest as a significant issue through the introduction of infotainment and camera-based Advanced Driver Assistance Systems (ADAS). These applications notably require more bandwidth than traditional control applications, and as such, the technologies and techniques used on current networks are insufficient for the needs of a next generation in-vehicle network architecture.

1.4.1 Open Systems Interconnection

The **Open System Interconnection** (OSI) is a standard for computer networks promoted by the *International Organization for Standardization* (ISO) which defines the logical structure of the network. It consists of "an architecture for open systems interconnection which could serve as a framework for the definition of standard protocols , layered architecture comprising **seven layers**" [18], each one specifying a certain aspect:

1. **Application Layer**: This is the highest layer in the OSI Architecture. Protocols of this layer directly serve the final application user and they make the communication compliant with the application.
2. **Presentation layer**: It provides the set of services which may be selected by the Application Layer to enable it to interpret the meaning of the data exchanged. Thanks to the

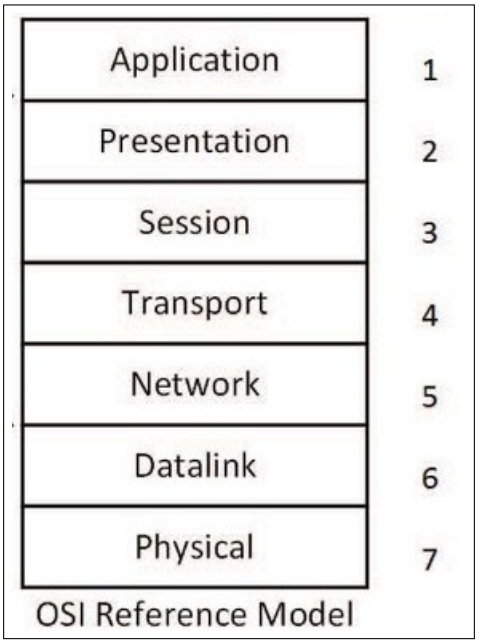

Figure 1.10. Open System Interconnection model Scheme

presentation layer, applications in an Open Systems Interconnection environment can communicate without unacceptable costs in interface variability, transformations, or application modification.

3. **Session Layer**: The purpose of the Session Layer is to assist in the support of the interactions between cooperating presentation entities. Its utility is both the binding two presentation entities into a relationship and unbinding them, as well as the controlling of the data exchange, delimiting and synchronizing data operations between two presentation entities.

4. **Transport Layer**: The Transport Layer exists to provide an universal transport service in association with the underlying services provided by lower layers, in order to optimize the use of available communications at a minimum cost.

5. **Network Layer**: The Network Layer provides functional and procedural means to exchange network service data units between two transport entities over a network connection. It provides transport entities with independence from routing and switching considerations.

6. **Data Link Layer**: The purpose of the Data link Layer is to provide the functional and procedural means to establish, maintain, and release data links between network entities.

7. **Physical Layer**: The Physical Layer provides mechanical, electrical, functional, and procedural characteristics to establish, maintain, and release physical connections (e.g., data circuits) between data link entities.

Moreover, a layer is only able to interact with the layers right below and above and provides new service or improve functionality with respect to the previous one. The interaction takes place from the bottom to the top.
However, not all the layers are implemented by the several protocols.

1.5 Controller Area Network

The **Controller Area Network (CAN)** is "a serial communications protocol which efficiently supports distributed realtime control with a very high level of security" [14], released in 1986 by Robert Bosch GmbH.
This communication protocol is used for connecting Electronic Control Units (see 1.1), also known as *nodes*.
In order to achieve design transparency and implementation flexibility CAN has been subdivided into different layers according to the ISO/OSI Reference Model (see 1.4.1):

1.5.1 CAN Physical Layer

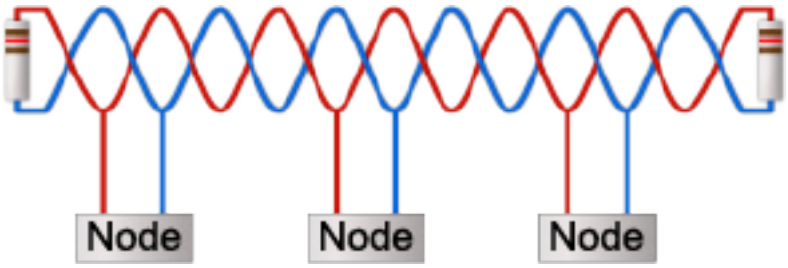

Figure 1.11. CAN physical layer

A twisted pair copper cable with common ground usually realizes the physical transmission. The choice of adopting two wires is to have the benefits of *differential signalling*:

- Cancellation of any unwanted interference that equally affect both wires induces common mode noise.
- Whenever the wires are affected by different noise, by twisting the pair of wires, the interference alternatively affects one wire and the other. This way, the noise becomes common mode and it is cancelled by differential signalling.

Most common is the high-speed transmission as standardized in ISO 11898-2:2003. It supports data-rates up to 1 Mbit/s.
The robustness of high-speed CAN networks is excellent. As said in **??**, in-vehicle networks often

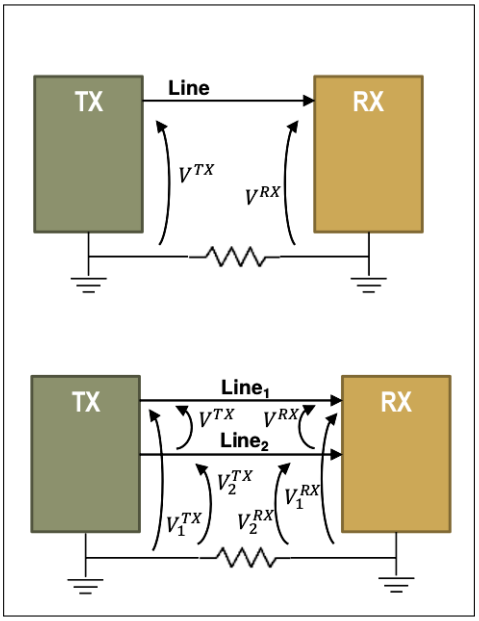

Figure 1.12. differential signalling

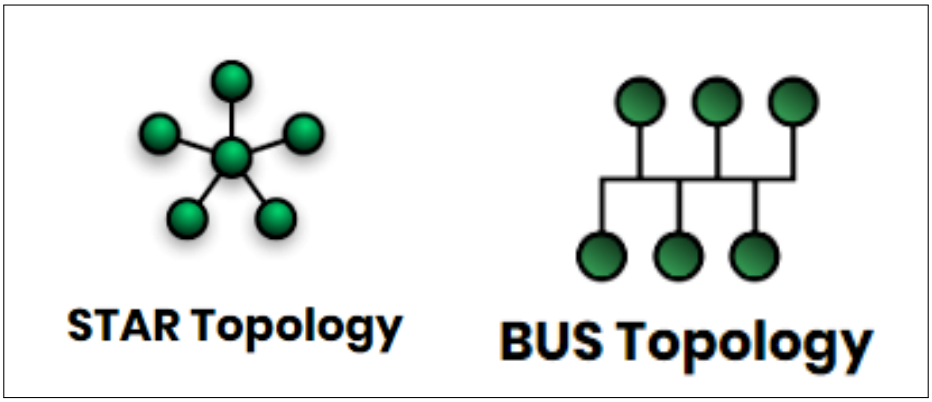

Figure 1.13. CAN topologies

use star-topologies, sometimes with multiple stars. In some applications, hybrid topologies are used, combining line and star. However, the most robust topology is a bus-line with very short stubs. For reference, see picture 1.13.

When in bus-line topology, the CAN messages are broadcasted. This means every node is able to consume any message produced by any other node in the CAN bus system.

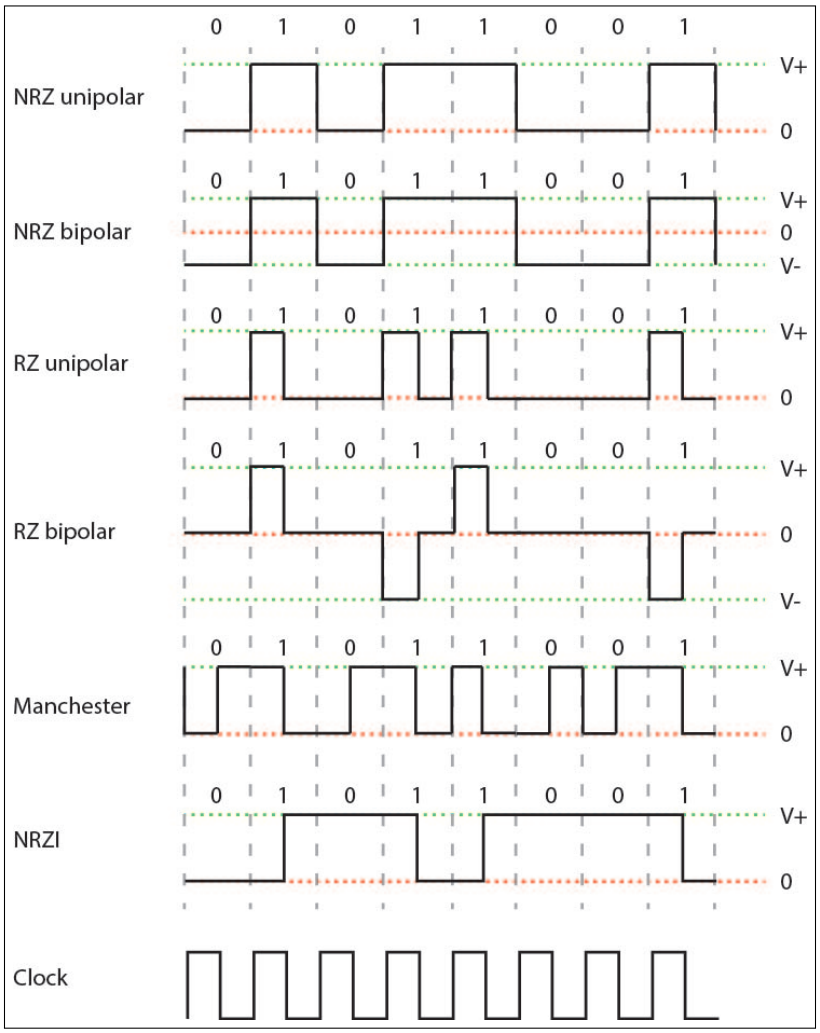

Figure 1.14. NRZ RZ and Manchester encodings, from [13]

Encoding and Electrical implementation

CAN uses **non-return-to-zero (NRZ) coding**. As we can see in 1.14, in contrast to the Manchester coding, not every bit contains a falling or a rising edge. During one bit, the voltage (0 V or +5 V resp. 3,3 V) is constant. The signal level can remain constant over a longer period of time if the transmitted bits have the same logical value. In order to avoid that the maximum permissible interval between two signal edges is not exceeded and that synchronization is not lost between transmitter and receiver(s), the solution of **bit-stuffing** is adopted: after five bits of the same value, the transmitting CAN controller automatically includes a bit of the opposite value. The receiving CAN nodes de-stuff the bit sequence, "meaning after five bits of the same value they automatically delete the following one" [**?**]. The CAN protocol transmits the serial bits as the

difference of voltage between the two twisted wires, called CANH and CANL.

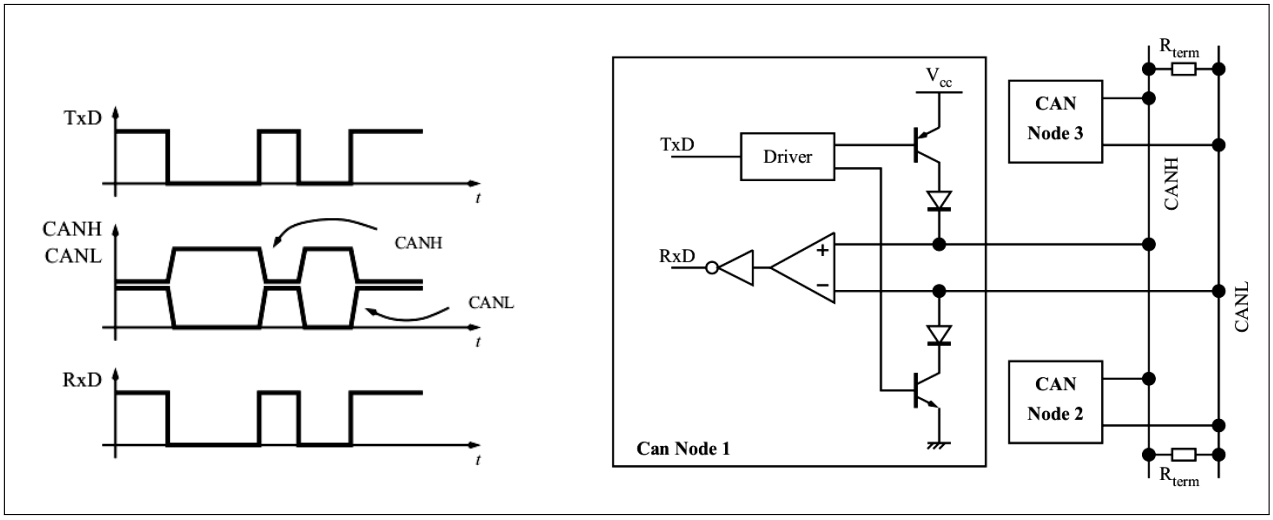

Figure 1.15. Voltage levels and electrical implementation of the CAN interface

Referring to figure 1.15, we can see that the two logical levels are:

- **Logic 0 - dominant bit**: CANH = high (5V), CANL = low (0V). Difference in reading is high, output of the amplifier is low.

- **Logic 1 - recessive bit**: CANH = float, CANL = float. Difference in reading is 0, output of the amplifier is high.

When none of the nodes connected to the bus is transmitting, a logic 1 can be read on the lines. As soon as a node starts sending any message, the recessive value of 1 is changed to the dominant value, which "wakes up" the other nodes.
Hence, combining the informations of NRZ encoding, bit stuffing and physical implementation, we can draw an example of CAN bit transmission, shown in figure 1.16:

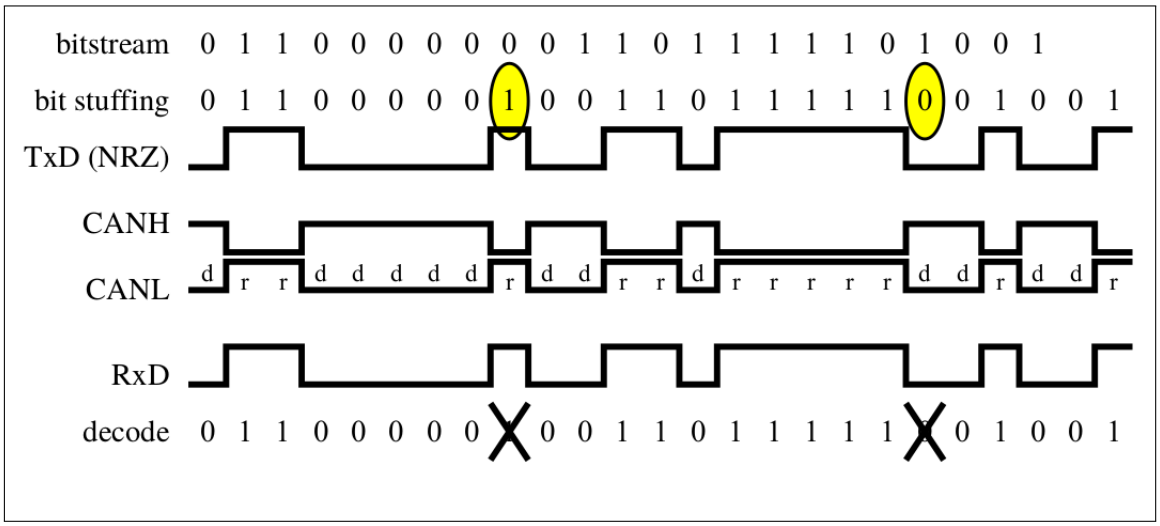

Figure 1.16. Example of CAN encoding

1.5.2 CAN Data Link Layer

Reading the official spefications of CAN bus [14], we know that information on the bus is sent in fixed format messages of different but limited length, called **frame**. When the bus is free any connected unit may start to transmit a new message.
In CAN systems, its nodes do not refer to the system configuration, which leads to the important consequences of:

- nodes can be added to the CAN network without any change in the software or hardware of any node.
- the content of the message can be named after an **identifier (ID)**, which does not point at the destination of the message but describes the message, its meaning and its sender. This way the nodes can activate a faster message filtering system, to decide whether the data is to be acted upon by them.
- Any number of nodes can receive and simultaneously act upon the same message.

Frames

The message transfer is manifested and controlled by four different frame types:

- *Data frame* carries data from a transmitter to the receivers,
- *Remote frame* is transmitted by a bus unit to request the transmission of the Data frame with the same identifier,

- *Error frame* is transmitted by any unit on detecting a bus error,
- *Overload frame* is used to provide for an extra delay between the preceding and the succeeding data or remote frames.

A data frame is composed of seven different bit fields: *Start of Frame (SOF), arbitration field, control field, data field, CRC field, ACK field, Enf of Frame (EOF)*. The latter description is on the standard frame, different from the extended frame. The two are very similar, but their differencies between do not touch this work.

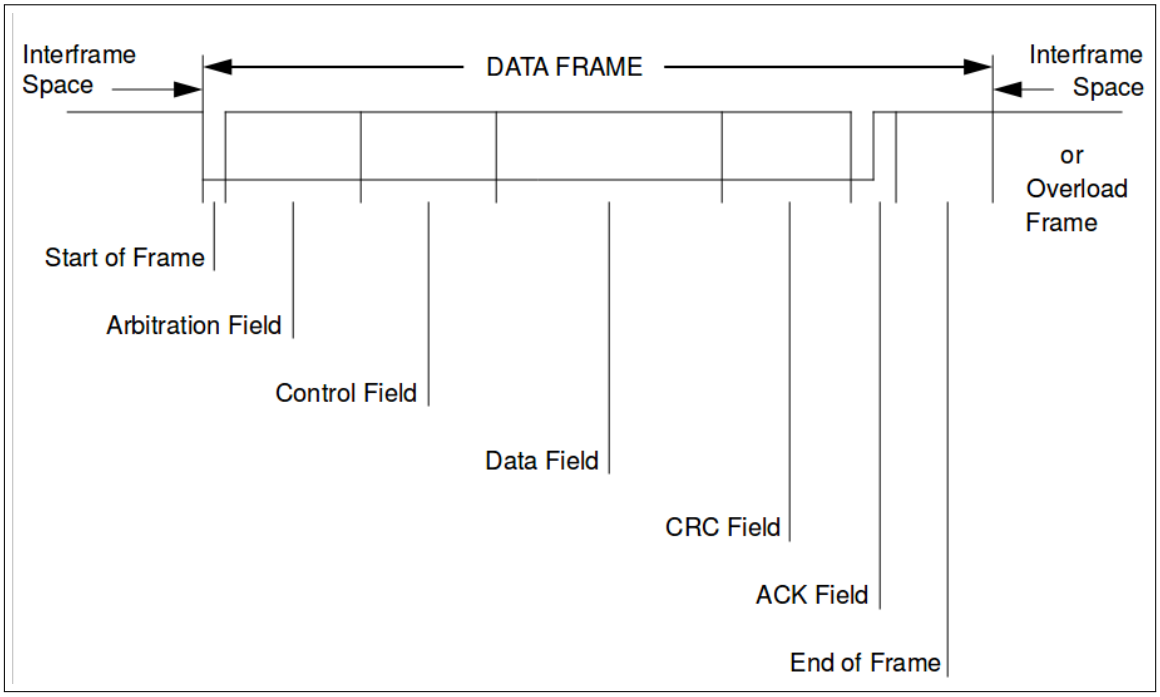

Figure 1.17. Data frame structure

- The **Start of Frame SOF** marks the beginning of the Data frame. A node is only allowed to start transmission when the bus is idle (CANH - CANL = 1, recessive bit).
- The **Arbitration field** consists of the *identifier* on 11 bits and the *remote trasnmission request bit (RTR bit)*, which consists of a recessive bit.
- **Control field**: on six bits, it consists of
  - the Identifier Extension Bit, a dominant bit to recognize that the frame is a standard frame,
  - a reserved bit r0,

  - the Data Length Code (DLC) on 4 bits, which expresses the length of the next frame fields, Data field.

- The **Data field** consists of the data to be transmitted within a data frame. It can contain from 0 to 8 bytes.

- The **CRC field** contains the CRC SEQUENCE followed by a CRC DELIMITER. We invite the reader to read the specifications [14] for a deeper understanding.

- The **ACK field** is two bits long and contains the ACK SLOT and the ACK DELIMITER. In the ACK FIELD the transmitting node sends two 'recessive' bits. A receiver node which has received a valid message correctly, reports this to the TRANSMITTER by sending a 'dominant' bit during the ACK SLOT (it sends 'ACK').

- The **End of frame EOF** consists of seven 'recessive' bits which delimit the frame just transmitted.

1.6 Automotive Cybersecurity

Autonomous driving, connectivity, and other initiatives are making automotive software increasingly complex. A modern high-end car features around 100 million lines of code, and this number is planned to grow to 200-300 millions in the near future [5].

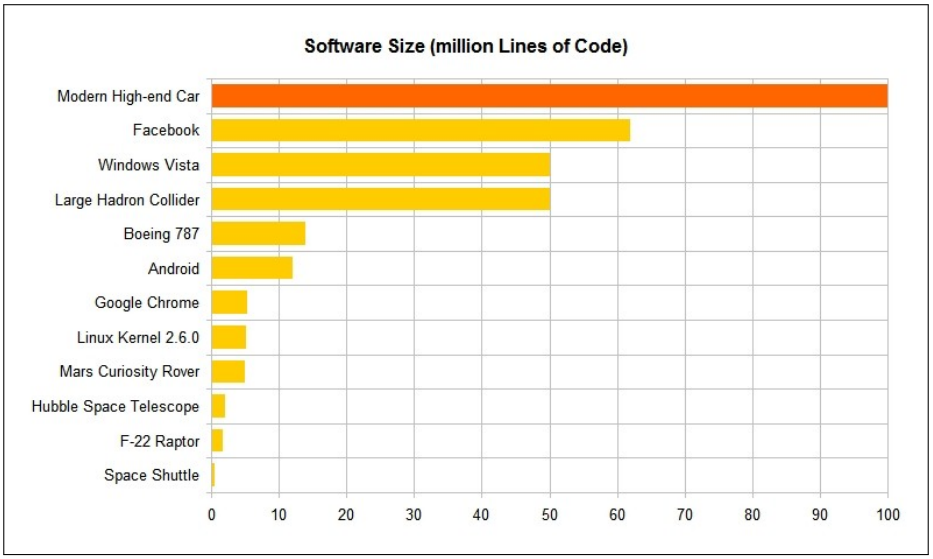

Figure 1.18. Software size comparison

An impressing comparison can be found in [6] and shown in figure 1.18: cars feature is by far one of the biggest pieces of software in terms of size, surpassing the F-22 fighter jet, a Boeing 787 and even a cumbersome operating system such as Windows Vista.
On the other hand, connectivity and large code size tend to expose vehicles to hacking, raising safety issues. As a matter of fact, the reader may remember some famous hacker attacks done on modern connected vehicles, such as the one in 2013 by Charlie Miller and Chris Valasek on a Ford Escape [10] or the hack by remote of the UConnect infotainment system of a Jeep Cherokee in 2015 [9].
Automotive cybersecurity is emerging as the discipline resulting from the evolution of cyber physical systems and connectivity in modern vehicles. Digital modernization in connected cars is opening up new threat vectors that are capable of **critically affecting the functional safety of vehicles**.
Cybersecurity in this industry needs to protect vehicle components and vehicle-related data from illegal access, interception, interference and modification.

1.6.1 Automotive Cybersecurity in CAN

Today, signaling buses and ECUs are still not enough secured and a hacker can inject malicious messages to make a car potentially unsafe [4]. One might even say that without security, the notion of a connected autonomous car cannot be safely realized. No safety means no connected autonomous car industry.
If autonomous cars are the next big thing, then a **truly robust automotive security architecture for ECUs**, gateways, domain/area controllers, and their manufacturing systems is the thing that makes the next big thing possible. As for security, as the number of computing nodes in a car network grows, the ways in which these nodes can be attacked increases exponentially.
Therefore, it is not an overstatement to say that robust cryptographic security is the essential innovation of the automotive future.
As we will see in 4, thanks to its wider payload, up to 64 bytes, CAN-FD introduces an interesting feature linked to cybersecurity in ECUs communication: the MAC/ARC, which is an encrypted authentication system based on the algorithm AES-CMAC.

1.7 Company context

This work was conducted in collaboration with the french branch of Bosch GmbH.
In 1886, with the help of his associate Arnold Zähringer, Robert Bosch succeeds for the first time in the world to provide to a motor vehicle, a De Dion-Bouton tricycle, with a low power VCR.
The inventions occurred afterwards. Between the most relevant, it should be noted the first high

power VCR marketed since 1902, the first injection pumps for gasoline engine in 1927, the first household appliances in 1933 as well as the production of ABS series, the first anti-lock braking electronic control system, in 1978.
With 410,000 associates worldwide spread in 130 engineering locations and 460 subsidiaries worldwide, in 2018, Bosch Group reached 78.5 million euros sales revenue, whereof the 79% coming from the locations outside Germany.
Bosch is now the biggest European company not quoted on the stock exchange.
In particular, this project was developed in the Bosch site of Saint-Ouen in the team PS-EC/ECP21. The formal tutor was dr. Antoine Poitrenaud (Antoine.Poitrenaud@fr.bosch.com). Moreover, the student colaborated with the Communication Stack responsible for Renault project Luc Prampolini (Luc.Prampolini@fr.bosch.com) and the responsible for the Team Leader in Real-Time HIL Test Systems Development Arnaud Sarazin (Arnaud.Sarrazin@fr.bosch.com).

1.7.1 Bosch Group Activities

Robert Bosch S.S.S. includes four principal activities:

Mobility solutions

Its main areas of activity are injection technology and powertrain peripherals for internal combustion engines, powertrain electrification, steering systems, safety and driver-assistance systems, vehicle-to-vehicle and vehicle-to-infrastructure communication, technology for infotainment and technology and services for the automotive aftermarket.

Industrial technology

The sector includes the Drive and Control Technology division, which specializes in drive and control technology for movement in machines and systems. The second division, Packaging Technology, provides process and packaging solutions for the pharmaceuticals and foodstuffs industries.

Consumer Goods

Its Power Tools division is a supplier of power tools, power tool accessories, and measuring technology (i.e. hammer drills, cordless screwdrivers, jigsaws, gardening equipment). The Consumer Goods business sector also includes BSH Hausgeräte GmbH, which offers a broad range of modern, energy-efficient, and increasingly connected household appliances (i.e. washing machines, tumble dryers, refrigerators and freezers, stoves and ovens, etc.).

Energy and Building Technology

Bosch' Building Technologies division has two areas of business: the global product business for innovative security and communications solutions, and the regional integrator business. The latter offers solutions and customized services for building security, energy efficiency, and building automation in selected countries. (i.e. video-surveillance, intrusion-detection, fire-detection, and voice-alarm systems, etc.). The Thermotechnology division provides heating, air-conditioning systems and energy management for residential, commercial and industrial heating environments. The Bosch Global Service Solutions division offers supports for external companies for business processes and services, primarily for customers in the automotive, travel, logistics industries and in information and communications technology.

1.7.2 Key data

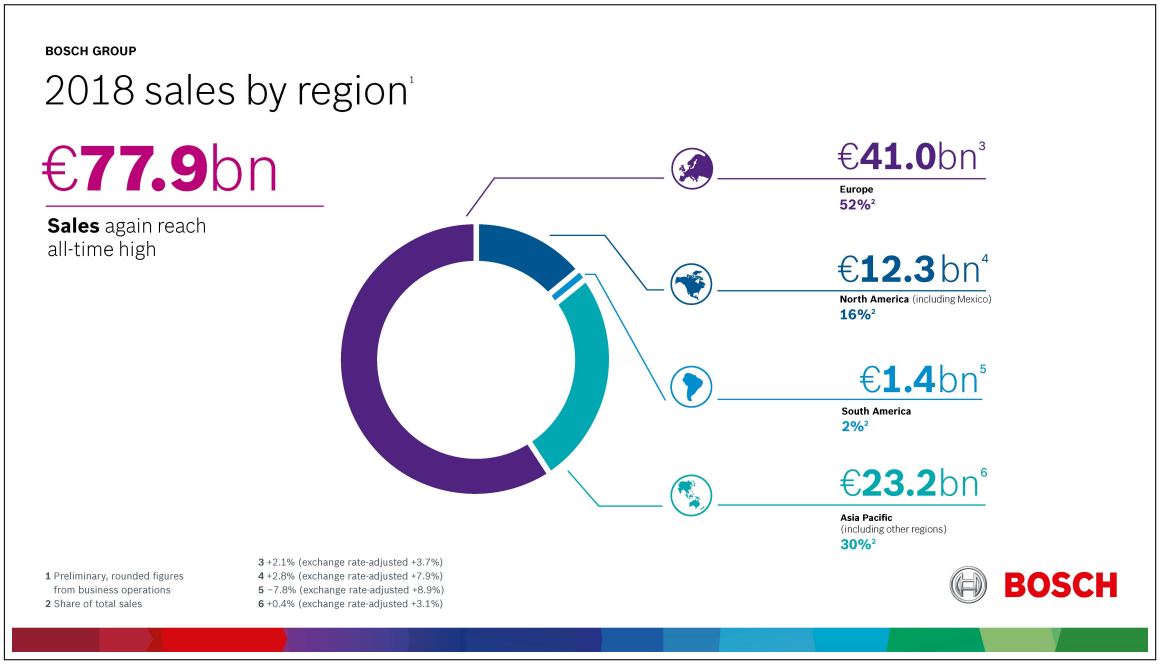

Figure 1.19. Bosch Froup 2018 business year

As we can see in figure 1.19, except from the sector of Consumer Goods, every development section increased its revenues in 2018.
Moreover, its remarkable effort in research and development is expressed in investments on new technology fields, such as:

- Automated driving: upfront investments worth 4 billion euros
- Electromobility: sales to reach 5 billion euros in 2025

- Artificial intelligence: 4,000 AI experts by 2021

1.7.3 Bosch France

France was the country were Bosch situated its first foreign site, opened in 1905 in Paris. With 23 sites, whereof 11 for Research & Development activities, each of the production sectors is represented in the Hexagon. In 2017, with a staff of approximately 7500 people, Bosch France realized a volume business of 3.2 billion of Euros in France.
It is worth of noticing that the index of equality between women and men for Bosch France reaches a score of 89/100.

1.8 Introduction to the subject and to the goals

Referring to figure 1.20, which shows the V cycle process model for SW product development in the Bosch company, this project lies in the context of *System Functionality Test*, in the red pane. The objective of the "System Functionality Test" (SFT) is to **verify the correct implementation of the functional requirements** of a "System Functionality" (SF), highlighted by the green pane. On the other hand, following the OSI (1.4.1), we can affirm that the the project aimed to cover the test of the *data link layer* of the communication protocol, the protocol layer in a program that handles **the moving of data into and out of a physical link** in a network.

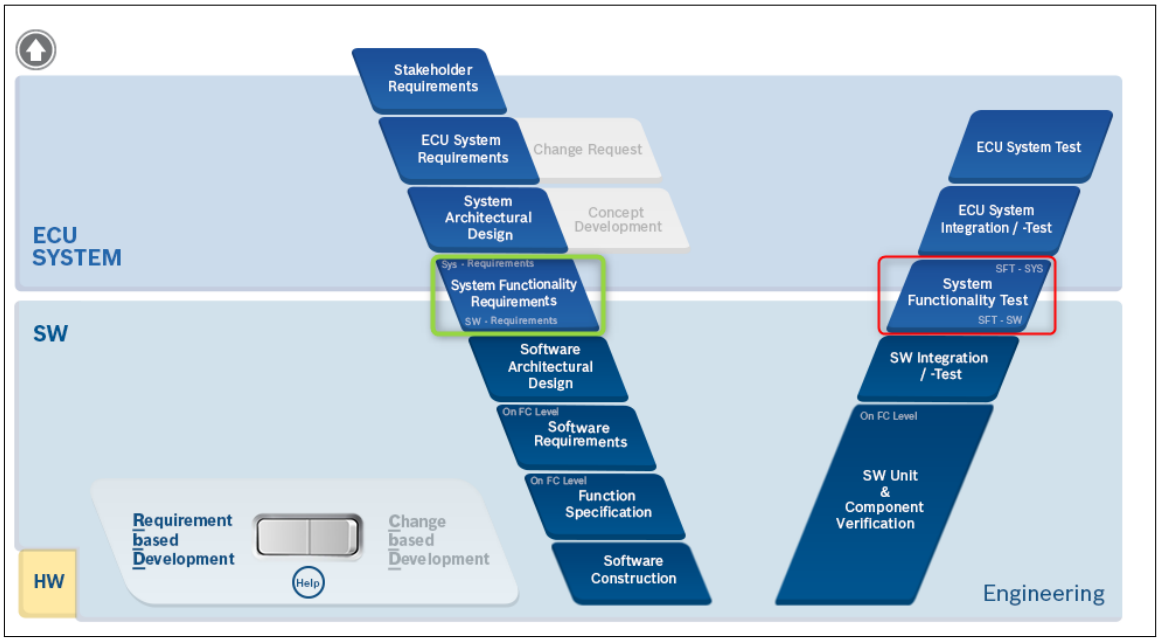

Figure 1.20. V cycle for the SW Product development

1.8.1 ComStack in Renault HEVC project

In particular, the goal for this project is to **test the coherence between the data present on the CAN bus and those in the Buffers of the CM module**.

In figure 1.21, we can see the software architecture of the ComStack (communication stack):

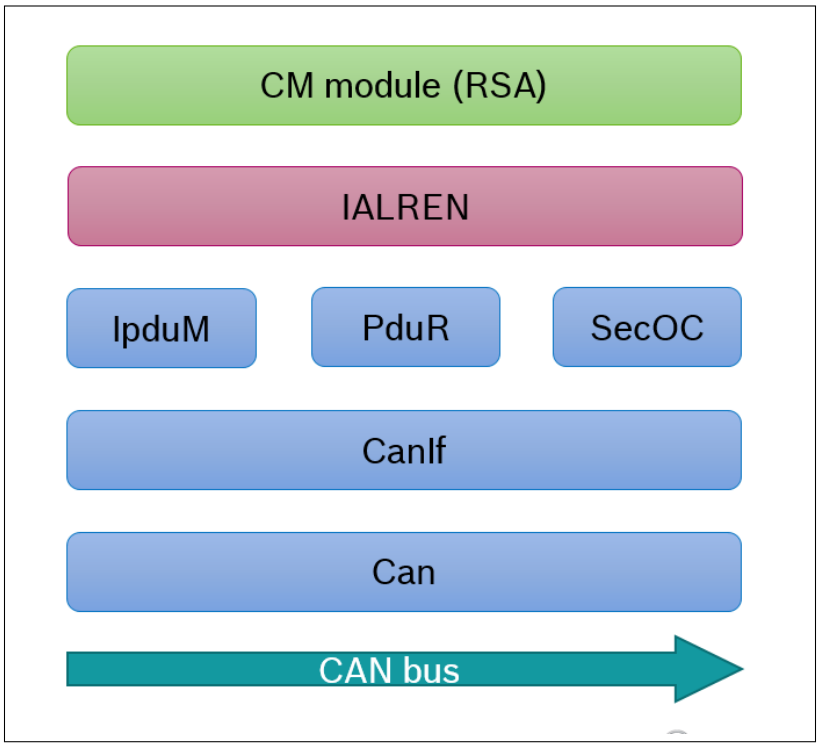

Figure 1.21. ComStack software architecture. In blue, the basic software Autosar; in red the customer specific team development. In Green, the software module.

- **Can**: CAN Controller Driver, it provides services for initiating transmissions and callback functions for notifying receive events, independently from the hardware
- **CanIf**: CAN Interface, it provides equal mechanism to access a CAN bus channel regardless of its location (microC internal or external). It abstracts from the location of CAN controllers (onchip/onboard), the ECU hardware layout and the number of CAN drivers.
- **PduR**: PDU Router, it provides services for routing of PDUs between the IpduM and the interfaces modules.
- **IpduM**: IPDU Multiplexer, it handles multiplexing of PDUs.
- **SecOC**: Secure Onboard Communication, aims for resource-efficient and practicable authentication mechanisms for critical data on the level of PDUs.

- **IALREN**: Interaction Layer (interface between Autosar ComStack and RSA CM module).
- **CM module**: RSA communication module.

The pilot project on which the first tests were developed is HEVC (electrical vehicle project). Nevertheless, these tests are suitable for every project exploiting the same software architecture CAN-FD Renault.

1.8.2 Project plan

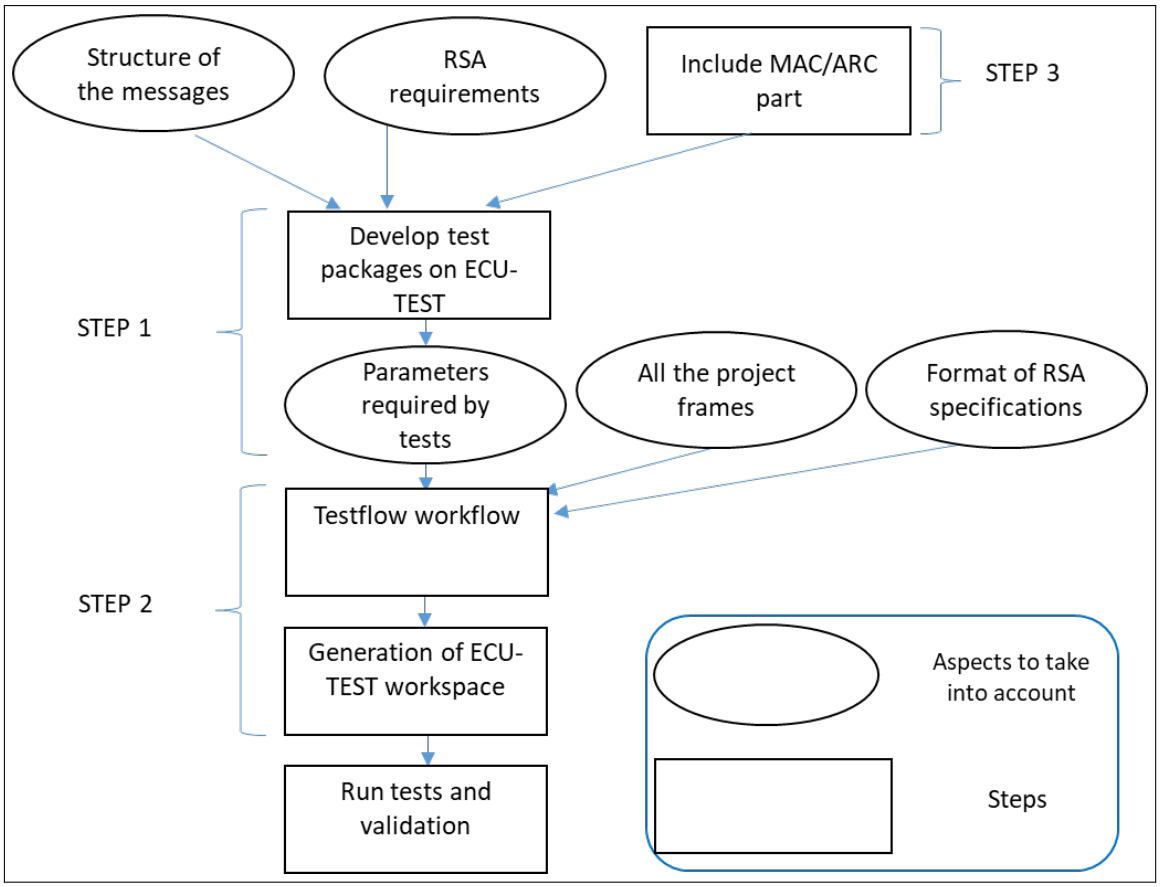

Figure 1.22. Project Plan

The main steps of this project are to:

1. Adapt the **already existing automated test** for the CAN frames to the CAN-FD PDUs and develop new automated tests for new objectives to cover (aspects to test). In this step, a particular care should be taken to the different message structure and the client requirements. In chapter 3, the reader can find a deep description of what above.

2. Automatically generate a test workspace with TestFlow, from the customer specifications. A brief description of this step will be provided in chapter 4.

3. Include in these test the MAC/ARC validation. This step is presented in chapter 5.

Moreover, a bonus step was previewed, involving the development of test plans for the validation of the gateway between different buses of an ECU under test. This step is described in chapter 6. For reference of the first three steps, see figure 1.22.

Chapter 2

Related work

State-of-art In the following chapter, related works associated to the item definition and validation are proposed.
The first section 2.1 shows the innovations of CAN-FD with respect to CAN protocol. These need to be fully described in order to understand the work done in the first step of this project, 3.
In the second section 2.2, the reader can find an introduction to the tools used in this project to develop, run and automatically generate the complete test plan, goal of the whole work. In the third section 2.3, we can find a brief overview of the main description file of the architecture under test.

2.1 CAN-FD

Today CAN(1.5) is still the most commonly used network in the automotive area, but the latest vehicle systems with high data rates require the introduction of CAN FD (CAN with flexible data rate) as its successor. CAN-FD (CAN with Flexible Data-rate) was introduced by Bosch in 2012 to overcome the Classical CAN's bit rate limitation to 1 Mbps and to expand the number of data bytes per CAN frame from up to 8 to up to 64, thereby closing the gap between Classical CAN and other protocols, as explained in 1.4.
In general, the idea is simple: when just one node is transmitting, the bit-rate can be increased, because no nodes need to be synchronized. Of course, before the transmission of the ACK slot bit, the nodes need to be re-synchronized.

2.1.1 CAN FD data field

CAN-FD, CAN Flexible Data-rate, is an extension to the original CAN bus protocol. Its primary message format is the Frame. The main difference was dealt with during this project was the

different frame format between CAN and CAN FD.

As we can read in [17], to get a better grouping of signals, which are related to each other, AUTOSAR (see 1.3) defined a new abstraction between signal and the frame. Signals are not mapped directly on a frame any more but to a **PDU(Protocol Data Unit)**. Later the PDU is mapped to a data link frame.
This allows using the same PDU in different frames and to propagate the PDU on different networks. Classic CAN does not really make use of this concept due to its limited payload of 8 bytes. AUTOSAR just defines a classical CAN Frame as a single PDU. The CAN-ID defines the content of the frame.

Container Frame

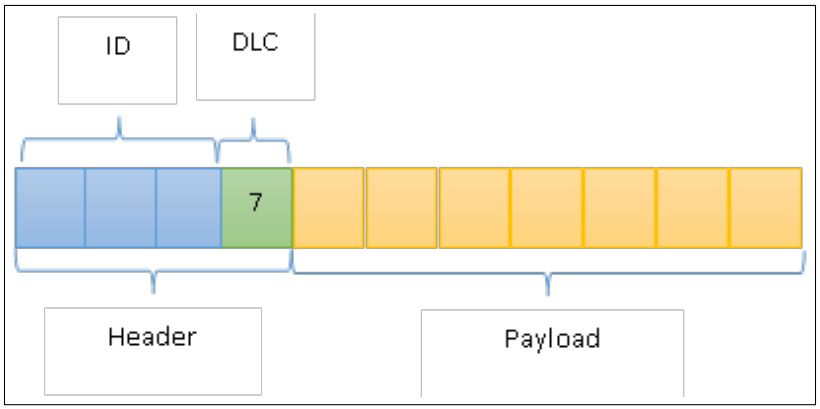

Figure 2.1. Contained PDU

Having the higher payload of CAN FD Multi-PDU-to-Frame Mapping becomes meaningful. The motivation is a dynamic construction of the frame during runtime. This means the position of a PDU within a frame is not static. Also the length of a PDU can be dynamic. Therefore a **header** identifying the PDU is required to enable the receiver to extract the single PDUs. This header consists of an ID and a DLC (data length code). In this work, we will call this data unit **contained PDU**. Its structure is shown in figure 2.1.
When mapping several contained PDU, we have the structure shown in figure 2.2, which, during this work, will be called **Container Frame**.

Regular frame

Another kind of frame is present in the CAN FD transmissions, called **Regular frame**: only one PDU is present and its ID is equal to the one of the frame. Therefore, the PDU header is not

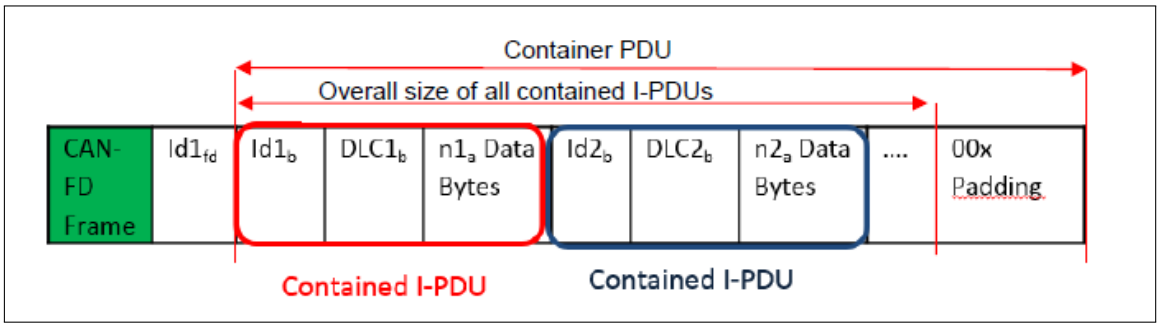

Figure 2.2. Container Frame

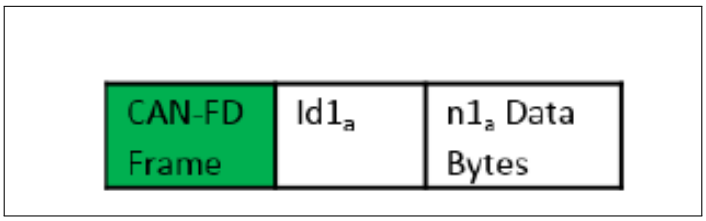

Figure 2.3. Regular frame

needed. For reference, see figure 2.3.

Secured frame

A Container frame in CAN-FD can be secured by the **MAC/ARC** authentication system, where MAC stands for *Message authentication code* and ARC for *Anti-Replay Counter*. In this case, the frame is called **Secured Frame**. Its structure is shown in figure 2.4.
In its structure, after the PDUs and the Padding, we have ARCLen bytes for the ARC and MACLen bytes for the MAC.
Further information about this security algorithm will be given in the section 5.1.

Pros and cons of the introduction of PDUs

With the new Multi-PDU-to-Frame Mapping the communication gets more independent of the network design. Hence the sender can be moved within the vehicle without modifying any receiving node. Moreover, unknown PDUs are skipped by the receiving node. Due to this, PDUs can be added to the vehicle without the need to update not affected ECUs, improving the reusability and flexibility of the system.
On the other hand, we can say that the drawbacks are higher CPU load in each ECU. The reason for this is that each node needs not only to receive frames but it also has to analyze them whether relevant PDUs are contained. As a consequence the vehicle network design needs to consider this

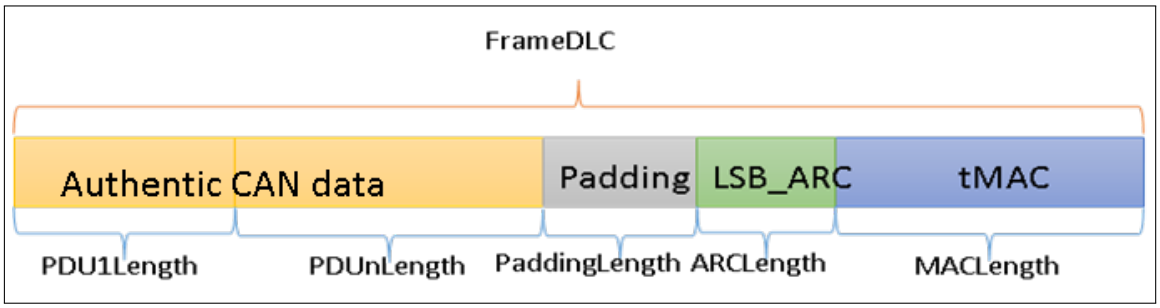

Figure 2.4. Secured frame. Authentic CAN DATA refers to the series of contained PDUs

aspect and keep irrelevant messages away from the ECUs. The domain architecture with its domain controller could take care to forward only relevant messages. For this, hardware filters cannot be used anymore to reduce the interrupt load.

Padding

In CAN FD, the Data field (1.5.2) of a frame has its length which has to fall in this set of values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48, 64, expressed in bytes. For this, it is necessary to insert in the frame some bytes of 0s after the PDUs in order to reach one of the length just mentioned. This mechanism is called “padding”. We will find further usages later in this dissertation.

2.2 Used tools

2.2.1 ECU-TEST

ECU-TEST is a tool for the validation of embedded systems in automotive environments. Its tasks are:

- Supports a broad range of test tools and test environments (SiL – HiL – vehicle)
- Standardized access to test tools
- Automation of distributed test environments
- Intuitive graphic user interface
- Generic test case description

This tool was chosen to develop and launch the tests for this project. The test to apply are developed in *packages*. Usually, for testing a single feature we develop a test package. This is composed of a **Test Case**, in figure 2.5, developed in:

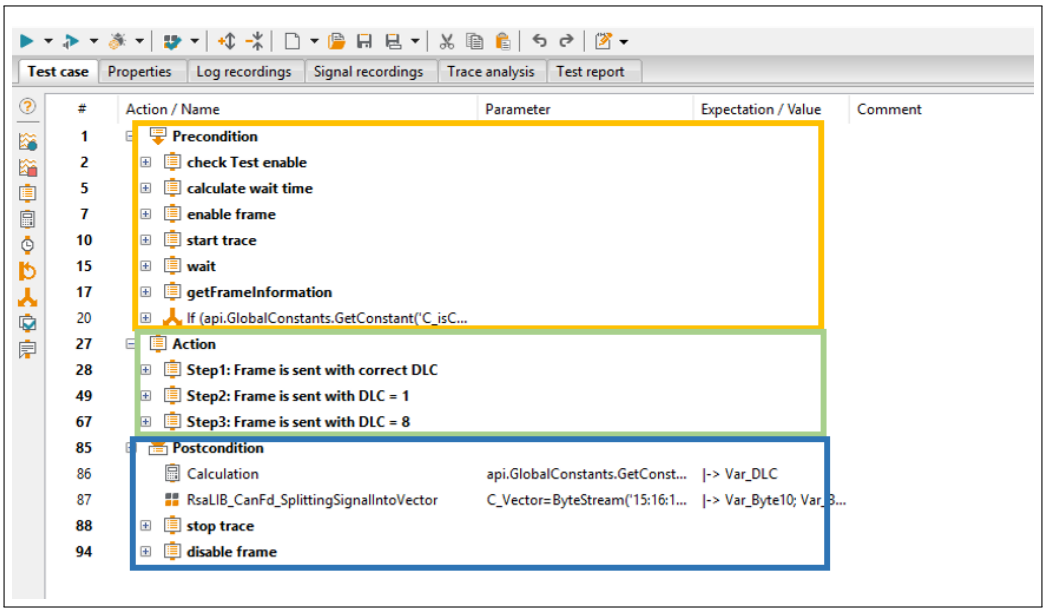

Figure 2.5. Test Case

1. Preconditions, in yellow, start of Signal recordings, synchronization, basic calculation on parameters and variables

2. Actions, in green, read and send messages on the bus

3. Post Conditions, in blue, end of the Signal Recordings.

After the execution of the test case, we find the **Trace Analysis**, as shown in figure 2.6. This is a section for plotting and processing the recorded signals, using *TraceStep Template*, python templates (called *UserPyModules*) or processing the signals. First in the synchronization part, in yellow, each recording is synchronized with the other. Then the signals are connected in signal connection, in blue. Finally, in episodes and trace steps, in red, the recorded signals are analyzed and processed.

2.2.2 Structure of the Test Bench

In order to test the proper behavior of the analyses performed on the channel, it was fundamental to install a proper and efficient test bench, where to write and read the channel and read the buffers.
Following the schema in figure 2.7, by the mean of the feature “activate PDU” in the ECU-TEST’ test cases, the PDUs are written and sent on the channel by the VectorHW VN1611. The latter is connected to the CAN port of the ECU. In order to read the signal present on the channel, the ETAS ES582.1 is present is connected to the laptop.

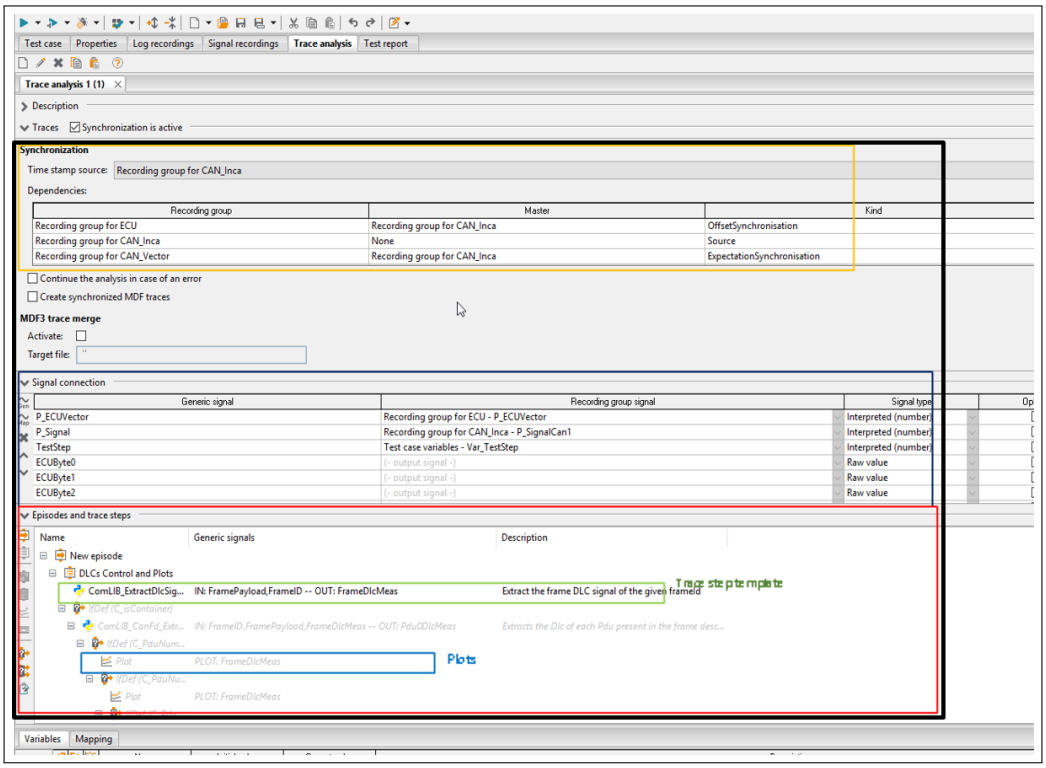

Figure 2.6. Trace Analysis

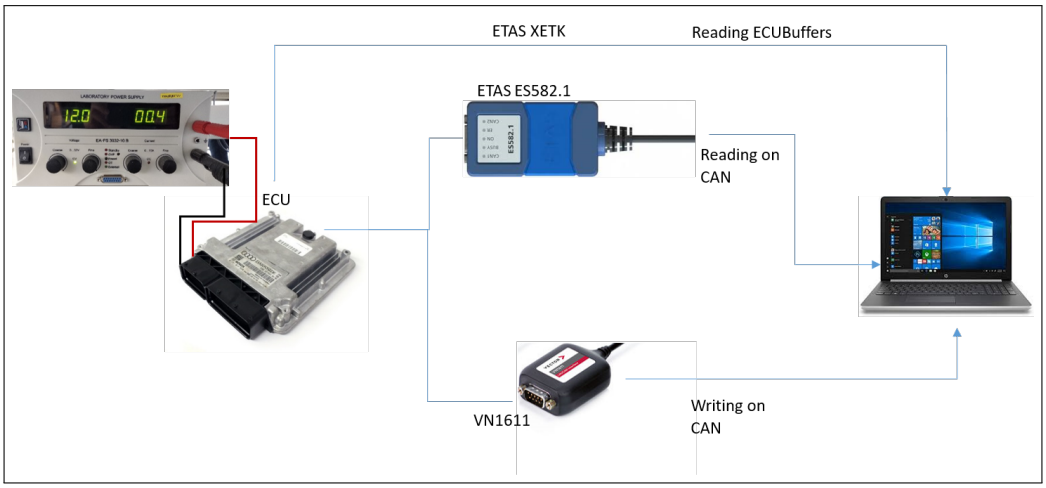

Figure 2.7. Test bench configuration

In conclusion, in order to read the ECUbuffers on the software INCA, an ETAS XETK cable is directly connected between the ECU and the computer.

2.2.3 Structure and role of the libraries

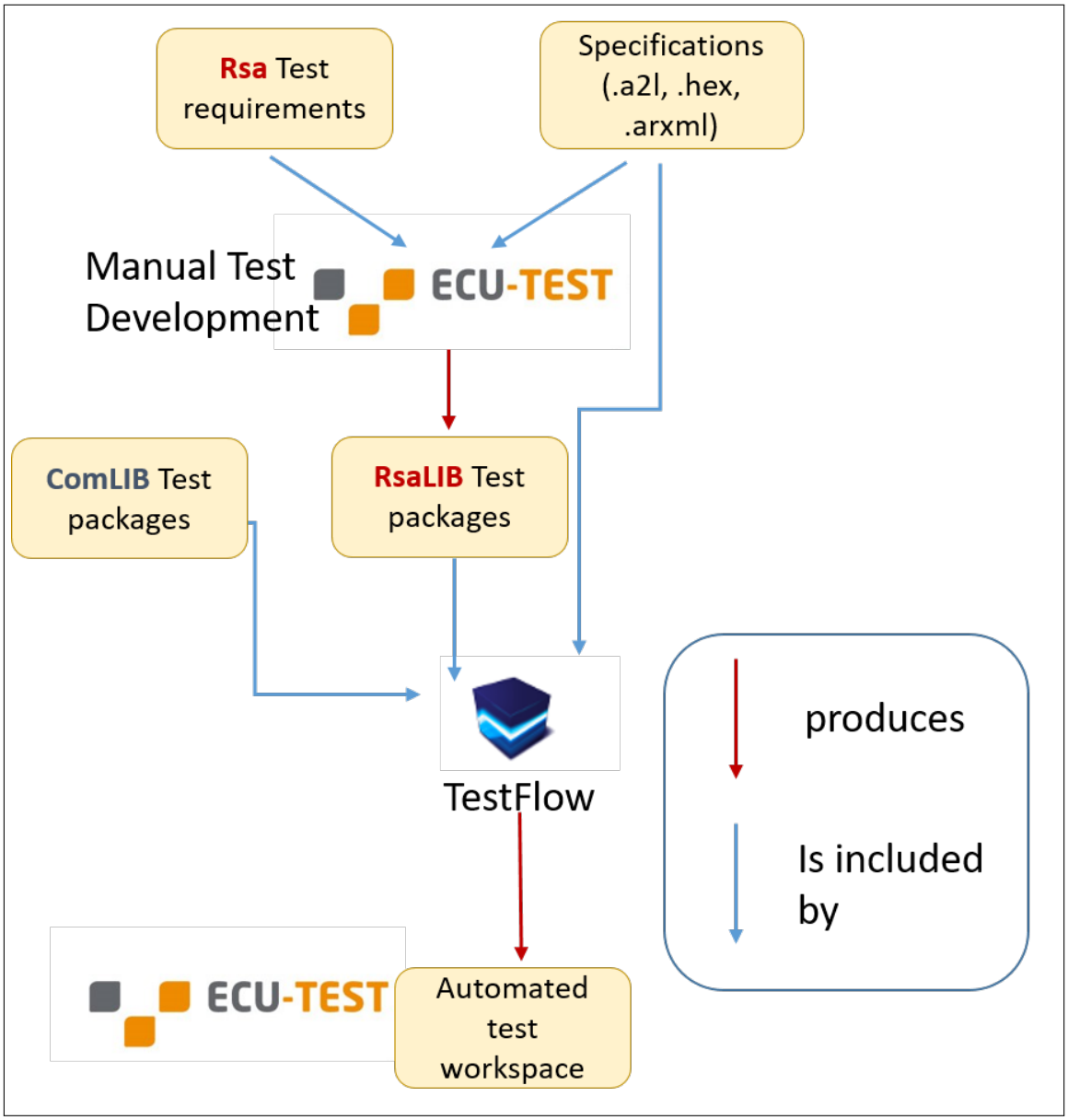

Figure 2.8. Step1 and Step2 workflow

The tests developed in this project are custom for the Basic Software Renault Nissan(RSA). These tests can rely on the platform library ComLIB, for when validating common properties to all the customer companies. In addition, those tests that validate specific properties belong to the custom library of the group: RsaLIB. The libraries are used in both of step 1 and step 2: during the development of the tests with the tool ECU-TEST and during the generation of the workspace

with the tool TestFlow(refer to 1.8).
In ECU-TEST, the libraries contain test packages, trace step templates and UserPyModules. Their function is to validate the properties specific to their company, in the case of customer library, or to the entire platform, ComLIB.
On the other hand, in TestFlow, the libraries contain the scripts to convert files, to map tests and to generate the automated test workspace.
For a further understanding of libraries' role in these first two steps of the project, refer to figure 2.8.

2.3 Input files

Here below, a short list of which input files are necessary to launch any of these test:

- .hex

  binary file, used to flash the ECU.

- .a2l

  description of the architecture of the ECUbuffers.

- .arxml

  AUTOSAR XML description file, which describes the frames, PDUs and signals.

2.3.1 ARXML File: plan dependencies

The development and the changes on the AUTOSAR XML file, described above, were not operated by the student, but by another Bosch associate[1].
Nevertheless, this kind of file is a fundamental part of the project and for its progress.
As an example, it was essential to wait for modifications of this file in order to have each signal belonging to a PDU mapped on an 8 bits long signal. This way, it was possible to compare more easily the bytes in the ECUbuffers - described in .a2l file - and the signal in the PDU (as shown in figure 3.10). For otherwise, a complex conversion for each of the signal of each PDU had to be implemented.
Similarly, in step 3 of the project (see section 1.8), the plan was to include in the tests the verification of the MAC/ARC security part. In order to correctly generate and analyze the MAC part, important parameters were needed in the .arxml, such as the secret encryption key.
In addition, it is relevant to say that in the file.arxml used in step 2, the presence of a security part

[1] Juilee Madhav Naik, Naik.JuileeMadhav@in.bosch.com

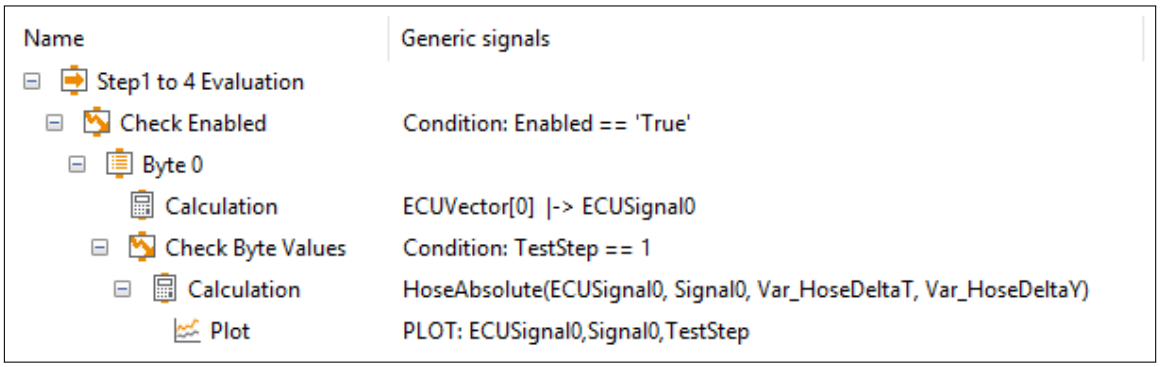

Figure 2.9. Comparison in the Rx_CheckSignalsConversion package between the signals and the ECUbuffer's bytes

was flagged. Still, this was done in an <ADMIN-DATA> part (shown in figure 2.10), which is the only portion of this kind of description file that is allowed not to follow the AUTOSAR system template.

Anyway, it was known by other RSA specifications that the lengths of MAC and ARC were respectively of 8 and 2 bytes.

```
<ADMIN-DATA>
  <SDGS>
    <SDG GID="Frame">
      <SD GID="GenMsgSendType">Periodic and Event</SD>
      <SD GID="Exclusion_time">4</SD>
      <SD GID="Optional_Frame">false</SD>
      <SD GID="GenMsgCycleTime">500</SD>
    </SDG>
    <SDG GID="Security">
      <SD GID="MAC">true</SD>
      <SD GID="AntiReplay">true</SD>
    </SDG>
  </SDGS>
</ADMIN-DATA>
```

Figure 2.10. .armxl used in step 2, all the data for security part

2.4 State of the art

Before the start of this project, the previous RSA projects of ECU did not include CAN-FD communication. Though, CAN communication was present and its validation was operated by

automated testing, with the tools ECU-TEST and INCA. These test plans were developed in the Bosch site of Saint-Ouen by the team of Arnaud Sarrazin and Aurélien Proust. Their work produced the CAN test libraries from which the student started to develop the test plan in step1, 3.

With the introduction of CAN-FD in the latest projects, the automated testing development team came up with the issue of the need of a new test architecture, that could also be compatible with the CAN-FD message format.

For the first RSA ECU project including CAN-FD, the first approach for CAN-FD validation was manual testing: starting from a test script, some replay blocks were generated and run on CANalyzer. Then, the data read in the ECUbuffers on INCA was compared with the CAN log. One can easily see how this first approach is time consuming and prone to human error.

By t

Chapter 3

STEP 1

In this chapter, there will be the presentation of the first step of the project, as presented in 1.8.2. Firstly, there will be described the RSA requirements (3.1 and 3.2), then the input files (2.3), the tools and the testbench used for this step (2.2) and, finally the solutions adopted for each of the requirements (3.3).

3.1 RX requirements

Here below, the list of the test requirements for the PDUs RECEIVED by the ECU:

3.1.1 Data

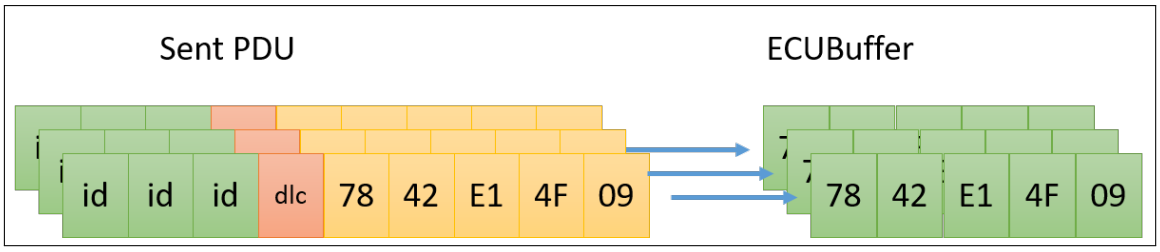

Figure 3.1. RX Data requirement satisfied

When a PDU is sent on a bus N times, the respective buffer RSA contains all the signals sent N times. For reference, figure 3.1

3.1.2 DLC_1

If a PDU is sent, not setting all of its signals, on the RSA buffer all the signals set will be found and in the not-set signals, we will find 0. For reference, figure 3.2

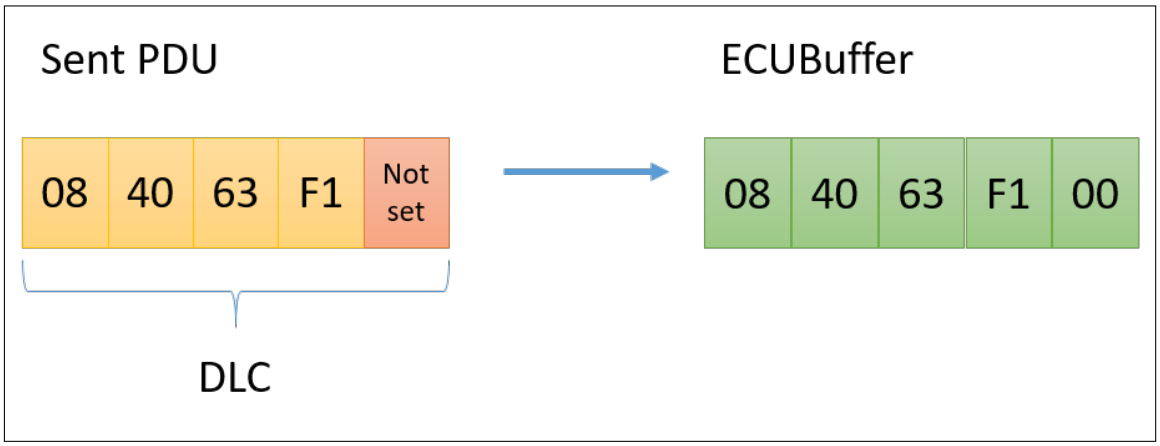

Figure 3.2. RX DLC_1 requirement satisfied

3.1.3 DLC_2

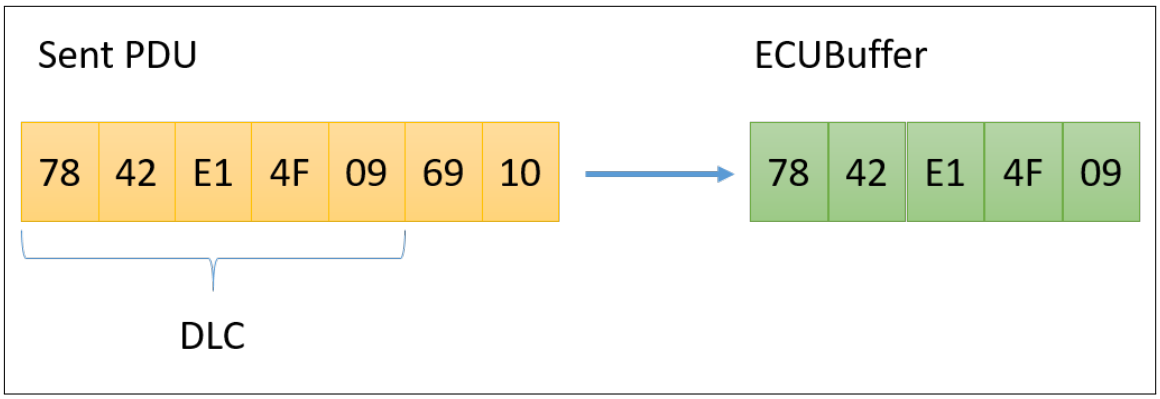

Figure 3.3. RX DLC_2 requirement satisfied

If a PDU is sent setting more bytes than its length, in the RSA buffer we will find only the bytes until its correct size. For reference, figure 3.3
Please note: *no test is required on the bytes after the DLC threshold.*

3.1.4 MessageCounter

For a PDU with a signal MessageCounter, none of reception occurrence shall be loss from RSA software. For reference, figure 3.4 and figure 3.5

3.1.5 Non-MessageCounter

For a PDU WITHOUT the signal MessageCounter, reception loss can occur once at a time and never consecutively.

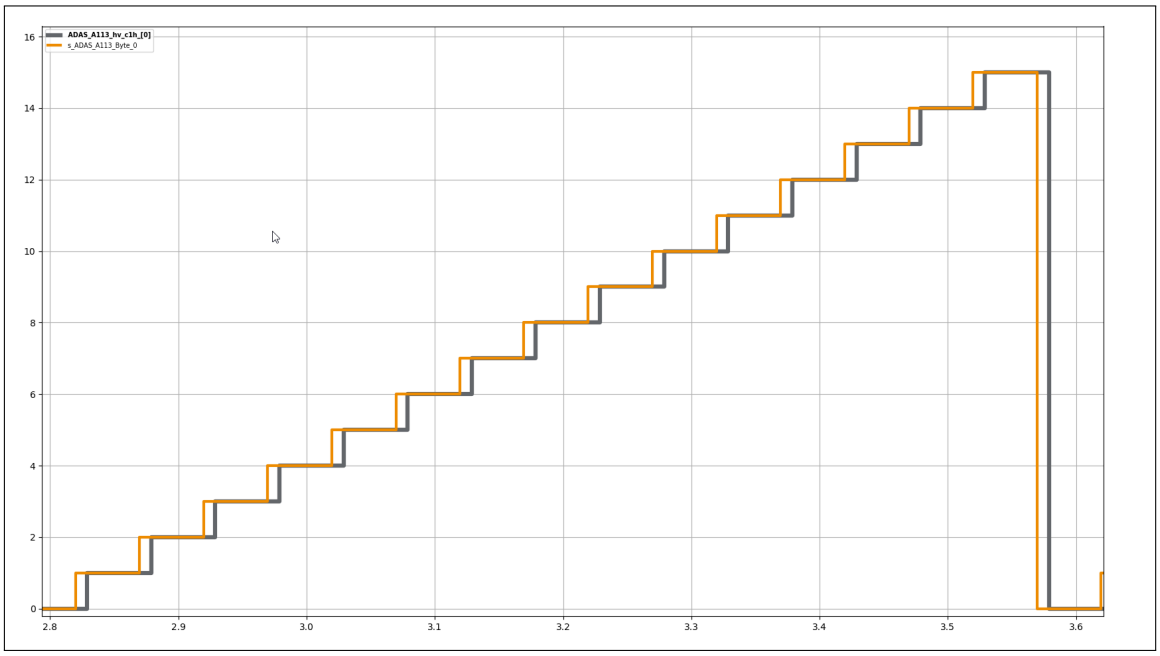

Figure 3.4. RX MessageCounter requirement satisfied

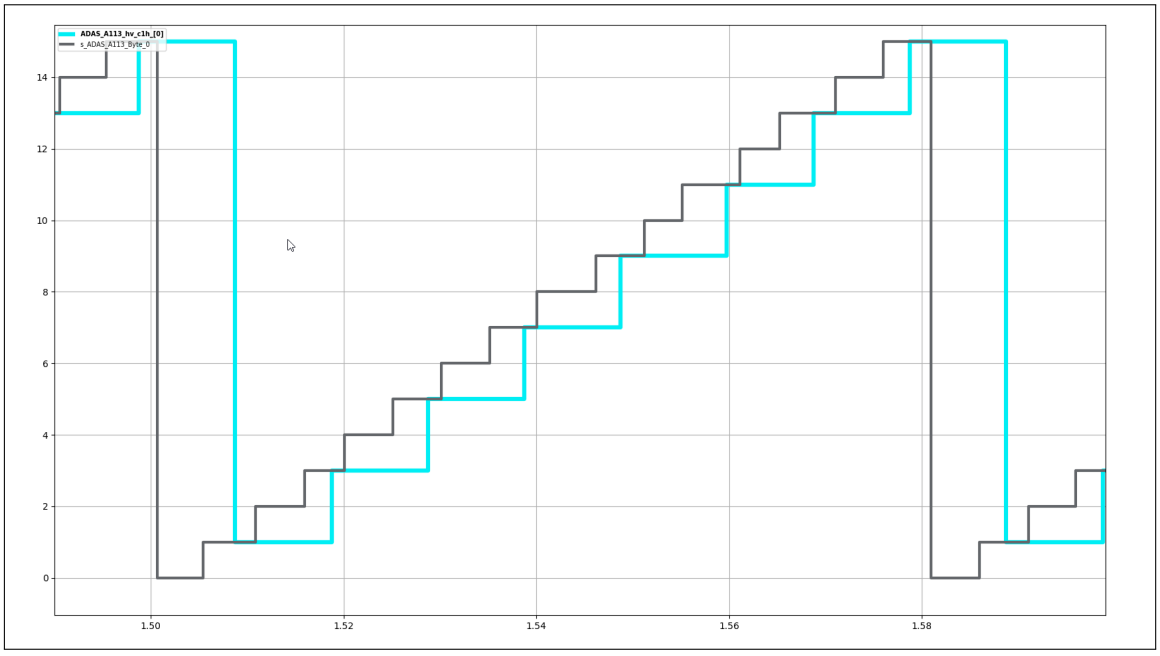

Figure 3.5. RX MessageCounter requirement not satisfied

3.2 TX Requirements

Here below the list of the test requirements for the PDUs SENT by the ECU:

3.2.1 Data+DLC

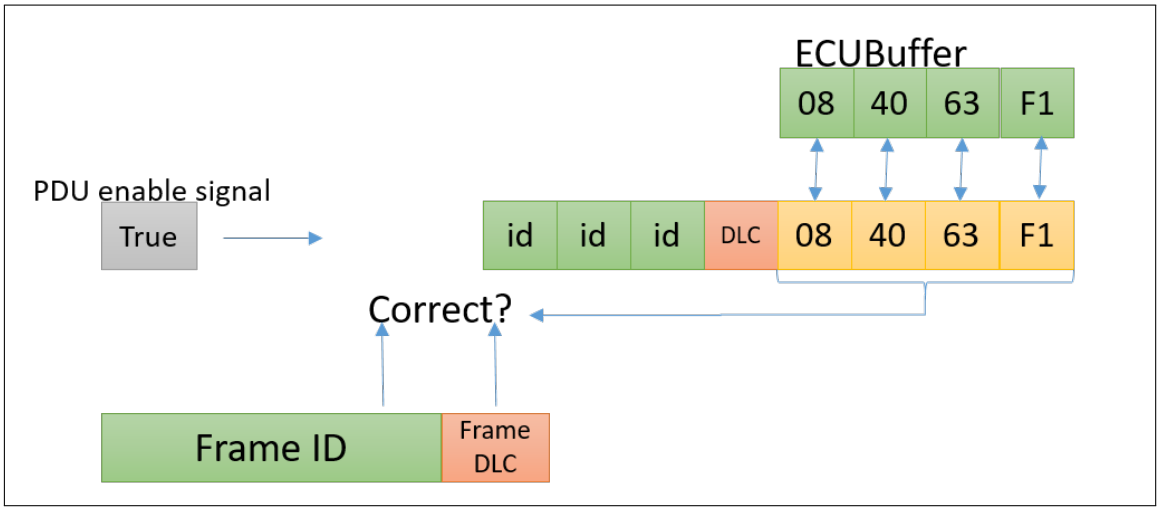

Figure 3.6. TX Data+DLC requirement satisfied

If PDU enable signal is set to active, then on the bus the good frame is found, with inside the correct PDU with the same data of the correspondent RSA buffer. The Frame ID, the Frame DLC and, if the frame is container, the PDU length have to be correct. For reference, figure 3.6.

3.2.2 Timing periodic

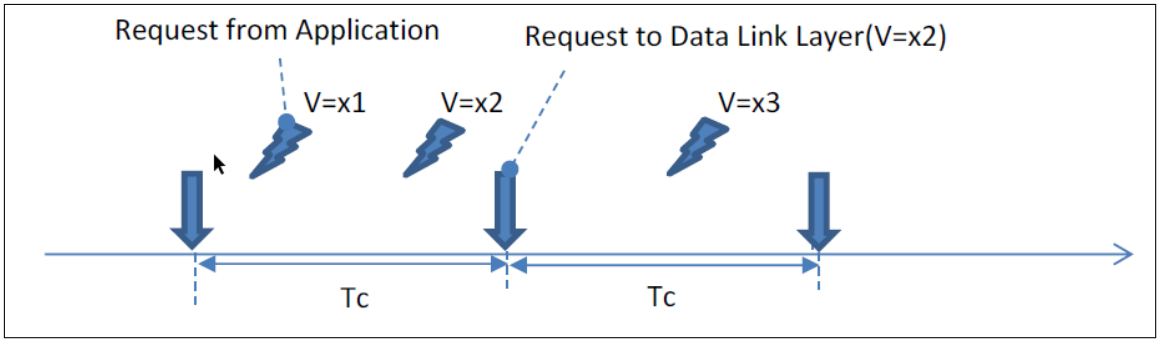

Figure 3.7. TX timing periodic and event triggered requirement satisfied

When analyzing a periodic with period P, the time difference between two sent frame Δt has to always respect:

$$P - 10\% <= \Delta t <= P + 10\%$$

For reference, see figure 3.7.

3.2.3 Timing periodic and event triggered

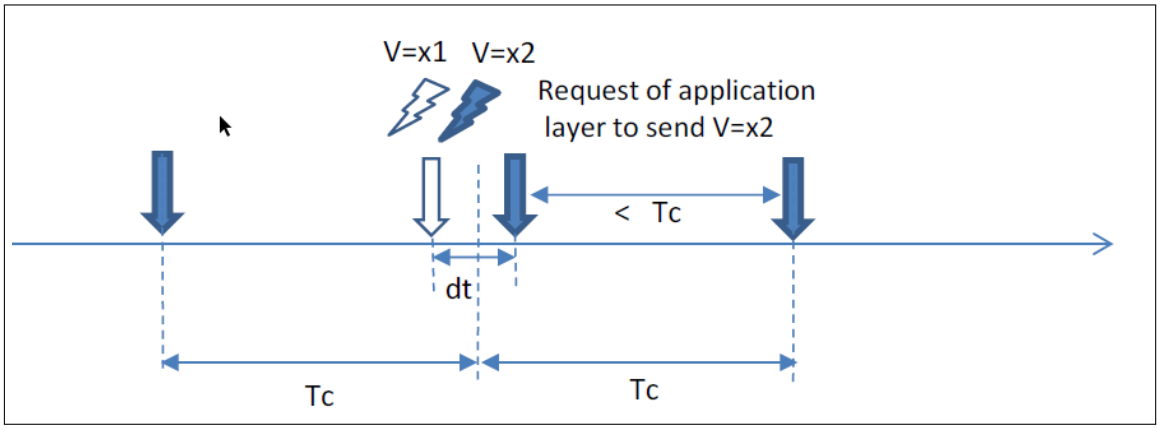

Figure 3.8. TX Timing periodic and event triggered requirement satisfied

When analyzing a periodic and event-triggered frame with period P and exclusion time t_{EXCL} (elapsed time after the last PDU sending), the time difference between two sent frame Δt has to always respect:

$$t_{EXCL} - 10\% <= \Delta t <= P + 10\%$$

3.3 Automated tests developing

All the test requests are described in the sections 3.1 and 3.2.

3.3.1 Rx: Data

This requirement, described in 3.1.1, was already implemented in a package called “RsaLIB_Can_Rx_CheckSignalCon
This package sends 8 PDUs with 8 different values on the bus. The heart of the package is in the trace analysis, shown in figure 3.10: here, in the "Calculation - HoseAbsolute" step, the first signal of the PDU is compared to the corresponding ECUVector, with a time tolerance and a value tolerance. In figure 3.9, the plot.
The intervention was a modification to a python script, Frame.py, where in the class “Calculate-Frame” a ByteStream is computed, including (see 2.2 and 2.1):

- the header (ID and DLC) for each PDU
- data to be transmitted for each PDU
- the MAC/ARC part, at the end of the sequence of PDUs, where present.

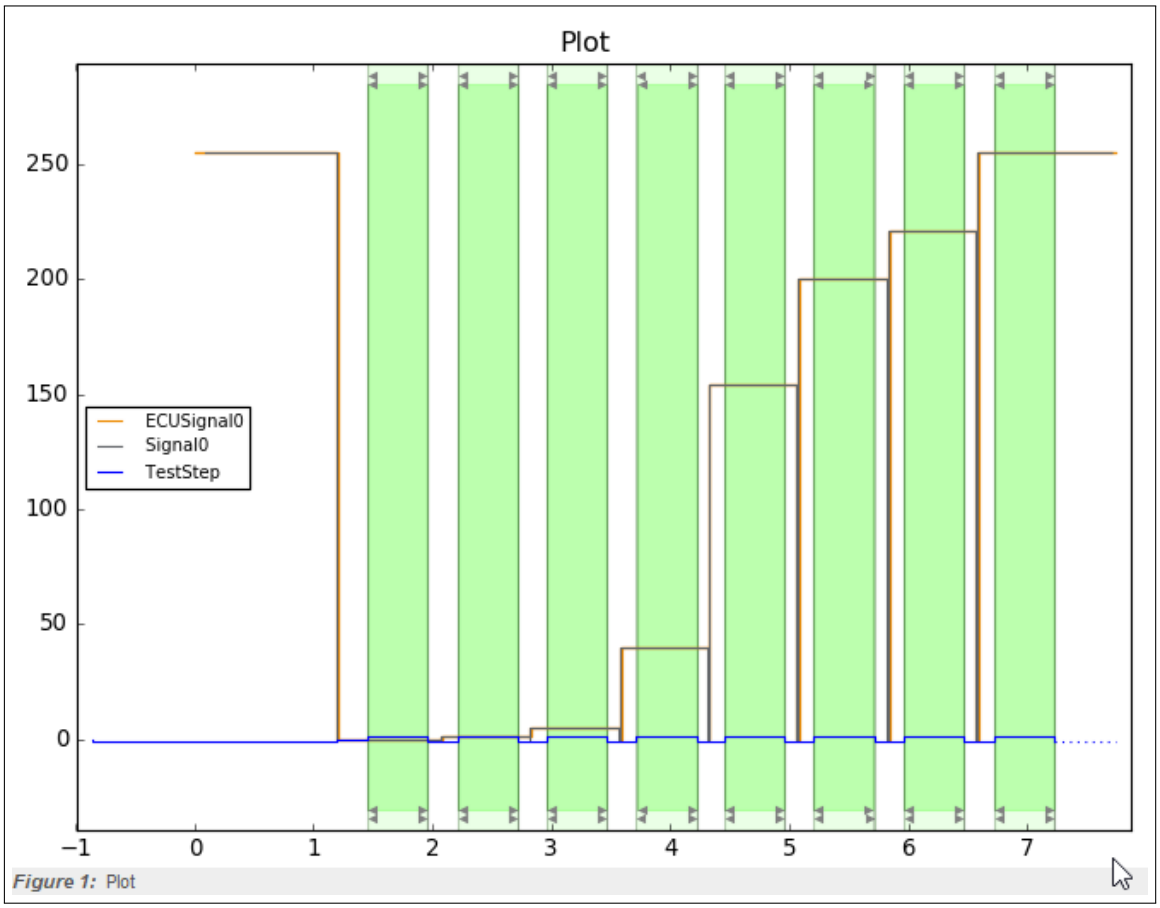

Figure 3.9. Plotting the behavior of a signal and of the corresponding ECUVector

Name	Generic signals
Step1 to 4 Evaluation	
Check Enabled	Condition: Enabled == 'True'
Byte 0	
Calculation	ECUVector[0] |-> ECUSignal0
Check Byte Values	Condition: TestStep == 1
Calculation	HoseAbsolute(ECUSignal0, Signal0, Var_HoseDeltaT, Var_HoseDeltaY)
Plot	PLOT: ECUSignal0,Signal0,TestStep

Figure 3.10. Comparison in the Rx_CheckSignalsConversion package between the signals and the ECUbuffer's bytes

The authentication part will be better described in the section 2.3.1. The goal of the modification was to include the header and the MAC/ARC part when the frame analyzed was of type container and/or secured.

3.3.2 Rx: DLC_1 and Rx: DLC_2

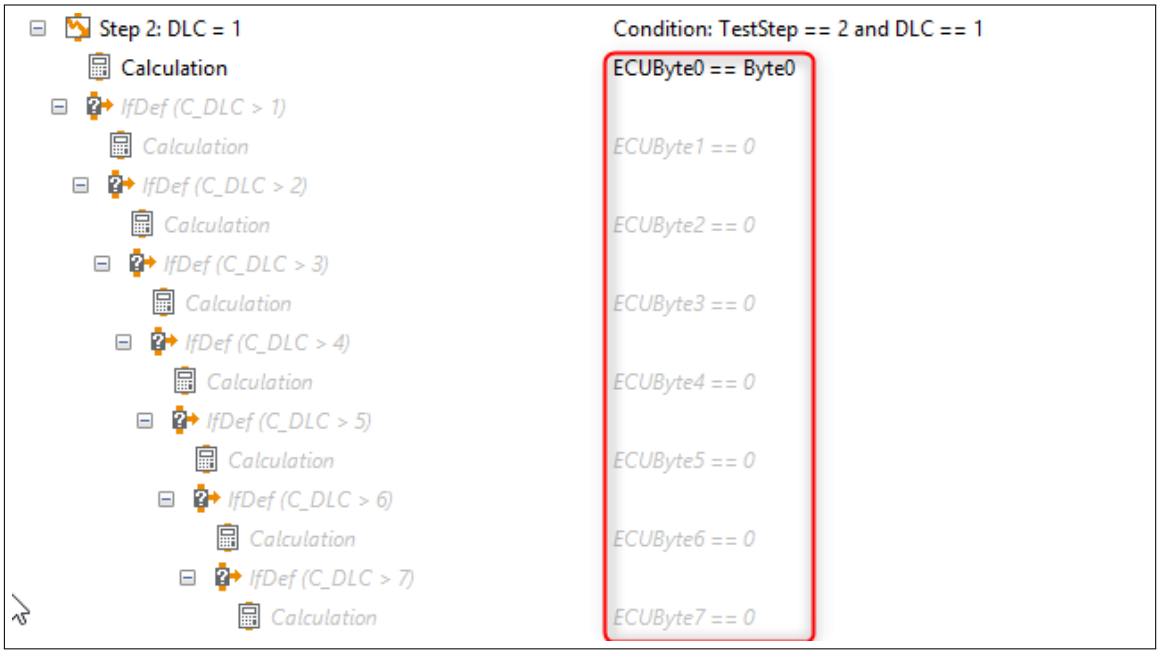

Figure 3.11. Control on latest bytes of ECUbuffer – DLC_1 implementation

These two requirements, described in 3.1.2 and 3.1.3, were added in the already existing package "RsaLIB_Can_CheckFrame_DLC". In the latter, the analyzed PDU is sent

- In STEP1, with its normal DLC,
- in STEP2, with DLC = 1, as asked by DLC_1 requirement,
- in STEP3, with DLC = 8, as asked by DLC_2 requirement.

In order to cover the requirements, in the tracestep analysis of this package a control on the latest bytes of the ECUbuffer was implemented:

- check of the last bytes to be set at 0, like shown in figure 3.11, for DLC_1 implementation
- the check of the correctly set bytes, like in figure 3.12 for DLC_2.

3.3.3 Rx: MessageCounter

This requirement, described in 3.1.4, was already implemented in the package "RsaLIB_Can_CheckFrameScheduling". In this package, the first signal of the PDU is incremented

- in STEP1, at the rate of its CycleTime,

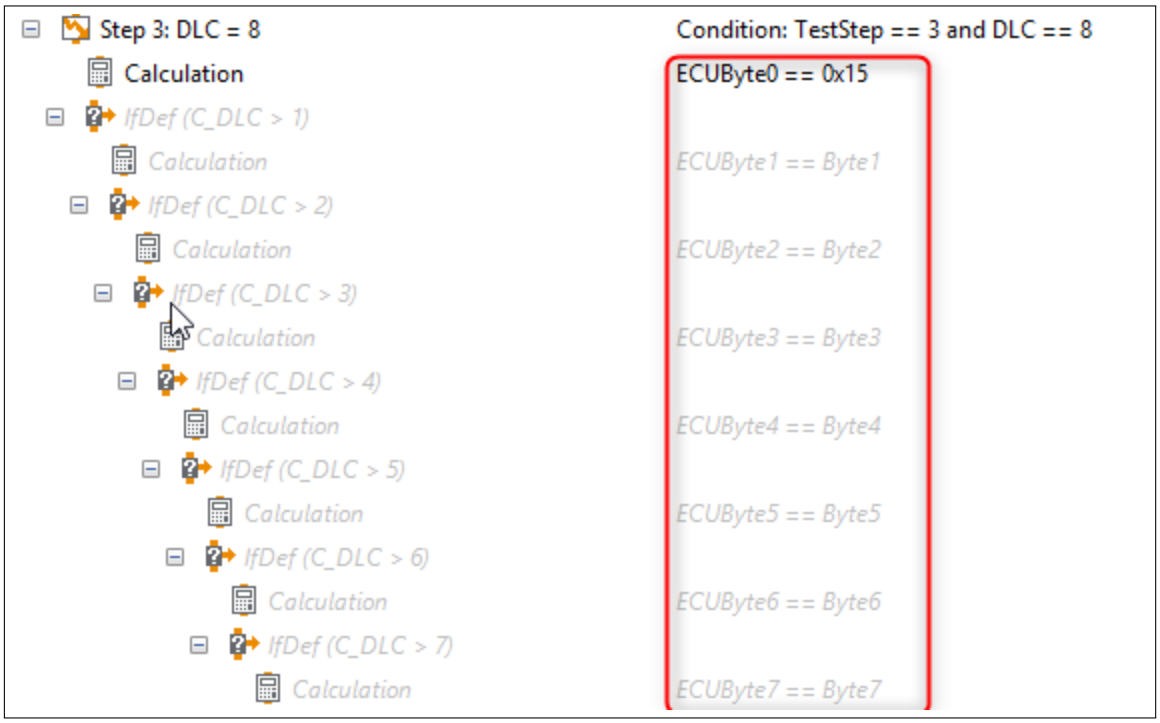

Figure 3.12. Control on the latest bytes of ECUbuffer - DLC_2 implementation

- in STEP2, at its half, CycleTime/2
- in STEP3, at its double, CycleTime*2.

For each of these steps, “ComLIB_CheckIncrementalSignal”, a trace step template verifies if its corresponding ECUbyte follows each of its variations. Moreover, various values of tolerances are taken into account such as:

- ValueTolerance – Absolute Value Tolerance on measured signal
- StepSize – expected increment from the “following” signal
- CycleTimeRelTolerance – relative tolerance regarding frame update in percentage
- MinDelayECUVsCan – minimum delay between CAN value update and SW update

In figure 3.14, a log from this trace step template is attached. As we can see, though the ECUSignal does not follow all of the variations of the signal, no failure is reported for this as the measurement is still inside the tolerance range.

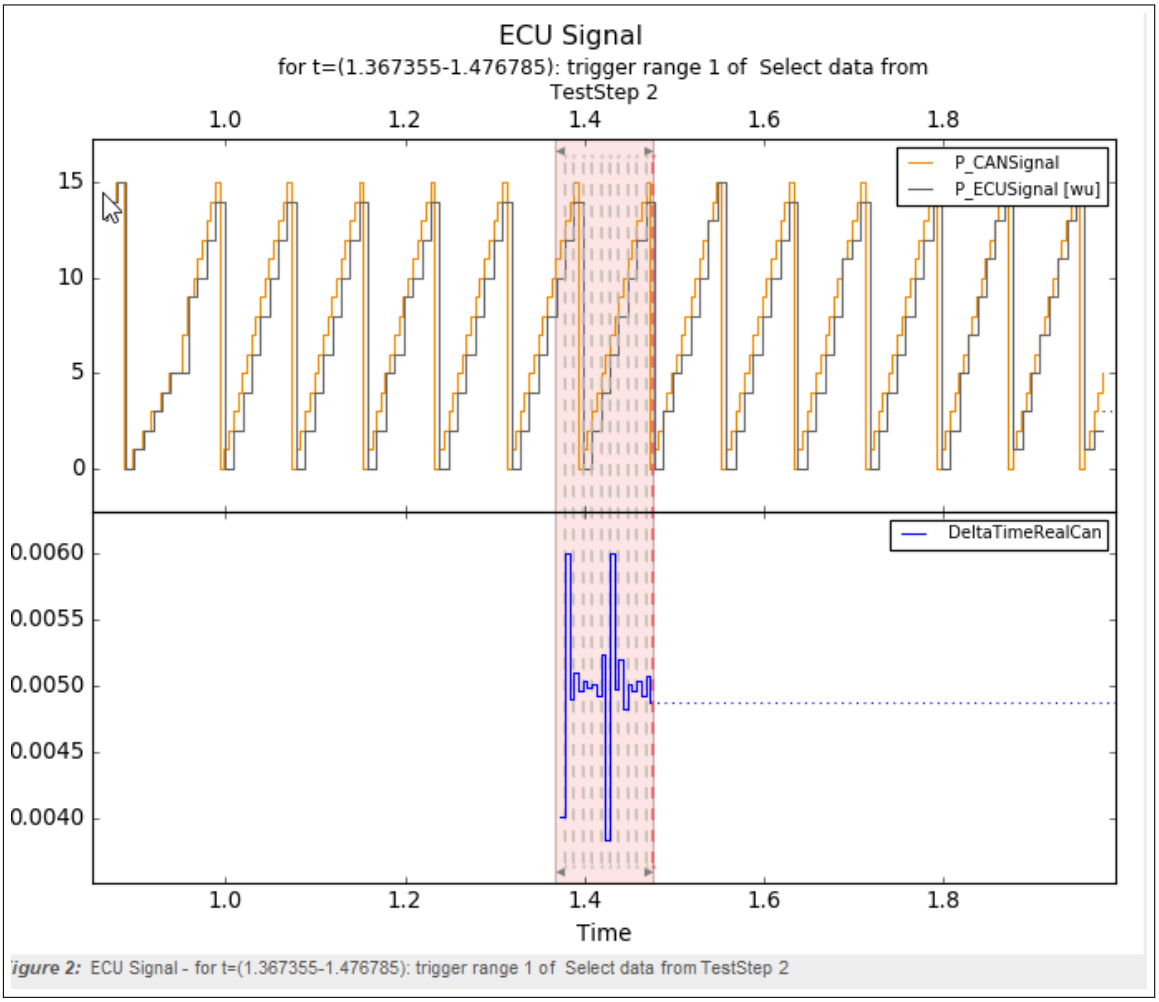

Figure 3.13. Plot of the timing analysis

Time	Duration	Result	message
1.367355 - 1.476785	0.109430	FAILED	trigger range 1
1.378296	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 11.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 11.0)
1.388265	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 13.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 13.0)
1.398282	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 15.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 15.0)
1.408299	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 1.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 1.0)
1.419280	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 3.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 3.0)
1.428286	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 5.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 5.0)
1.438255	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 7.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 7.0)
1.448272	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 9.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 9.0)
1.458289	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 11.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 11.0)
1.468307	-	NONE	Missed but Inside increment tolerance: Previous Counter Value: 13.0 with tolerance: 0.1 was missed by the ECU (Raw CAN Value: 13.0)
1.476301	-	FAILED	Missed variations over tolerance: 45%

Figure 3.14. Log of the timing analysis

3.3.4 Rx: Non- MessageCounter

This requirement, described in 3.1.5, was never implemented before. Indeed, in a first approach was to create another TraceStep Template to be included in the “RsaLIB_Can_CheckFrameScheduling”:

when the ECUbuffer could not follow two signal variations consecutively, a failure would be launched.
Lately, better analyzing the already existing script “Com-LIB_CheckIncrementalSignal”, it was possible to include this requirement in the existing tests by setting a parameter, stepsize = 2. Stepsize is the length of the expected increment by the “following” signal.
Moreover, a soil of missed frame was added, independent from the stepsize value. If this soil is overpassed (10%), a failure will occur, as in figure 3.14 .

3.3.5 Tx: Data+DLC

This requirement, described in 3.2.1, was partly implemented in the package “RsaLIB_Can_Tx_CheckSignalsWoStim” the ECU was activated and when the frame is active, each signal of its PDU is compared to the corresponding ECUByte.
A check on the FrameDLC and the length of the PDU was added in the TraceAnalysis, by the script “Com-LIB_CanFd_ExtractContainedPduLength”, developed by the student.
Here, starting from a payload recording, for every frame emission, each of these parameters is analyzed:

- FrameDLC,
- PDU’s DLC,
- PDU’s ID,
- Payload.

These parameters, in particular PDU’s DLC, were compared to the expected values, computed from the parameters in the arxml file.
Lately, this check was added also to the package “RsaLIB_CanFd_CheckFrameScheduling”.

3.3.6 Tx: timing periodic

This requirement, described in 3.2.2, was already implemented in a ComLIB package, “Com-LIB_CanFd_Tx_CheckPduEmission”. Here, the ECU is activated and a range of time necessary to collect 10 frames is waited. Finally, by the mean of the TraceStep Template “Com-LIB_CheckAllFramesPduCycleTime”, the time elapsed between the emission of the same frame is measured and it is compared to the period with a certain tolerance, for both upper and lower limit

$$P - 10\% <= \Delta T <= P + 10\%$$

.

3.3.7 Tx: timing periodic and event triggered

This requirement, described in 3.2.3, was never implemented in the in the existing packages or TraceStep Templates. However, it was sufficient to modify “ComLIB_CheckAllFramesPduCycleTime” to have an upper limit and lower limit.
Later on, with the help of the responsible for ComLIB tests and custom libraries , we realized that it was better to merge the latter two requirements into a unique package “ComLIB_CanFd_Tx_CheckPduEmission”.
Here, in the tracestep template “ComLIB_CheckAllFramesPduCycleTime” it was possible to implement a symmetric limit of time

$$P - 10\% <= \Delta T <= P + 10\%$$

as well as an asymmetric limit of time:

$$t_{\mathrm{EXCL}} - 10\% <= \Delta T <= P + 10\%$$

3.3.8 Resuming table

Requirement identifier	*Package*	*TraceStep Template or UserStepTemplate*
Rx: Data 3.1.1	RsaLIB-Can- Rx-CheckSignalConversion	Frame.py → CalculateFrame
Rx: DLC-1 3.1.2	RsaLIB-Can-CheckFrame-DLC	
Rx: DLC-2 3.1.3	RsaLIB-Can-CheckFrame-DLC	
Rx: MessageCounter 3.1.4	RsaLIB-Can-CheckFrameScheduling	ComLIB-CheckIncrementalSignal
Rx: Non-MessageCounter 3.1.5	RsaLIB_Can-CheckFrameScheduling	ComLIB-CheckIncrementalSignal
Tx: Data+DLC 3.2.1	RsaLIB-Can-Tx-CheckSignalsWoStim	ComLIB-CanFd-ExtractContainedPduLength
Tx: Timing periodic and event triggered 3.2.3	ComLIB-CanFd-Tx-CheckPduEmission	ComLIB-CheckAllFramesPduCycleTime
Tx: Timing periodic 3.2.2	ComLIB-CanFd-Tx-CheckPduEmission	ComLIB-CheckAllFramesPduCycleTime

Chapter 4

STEP 2

In this chapter, there will be the presentation of the second step of the project, as presented in 1.8.2.
Firstly, there will be in 4.1, a general presentation of the workflow and later on, the description of the work done on the generation scripts.

4.1 TestFlow workflow

After developing the tests and covering each of the test requirements, it was needed to fulfill the automated generation of a workspace, where to each PDU the proper test package are applied. This is done by TestFlow. Figure 4.1 represents the workflow adopted for this project.

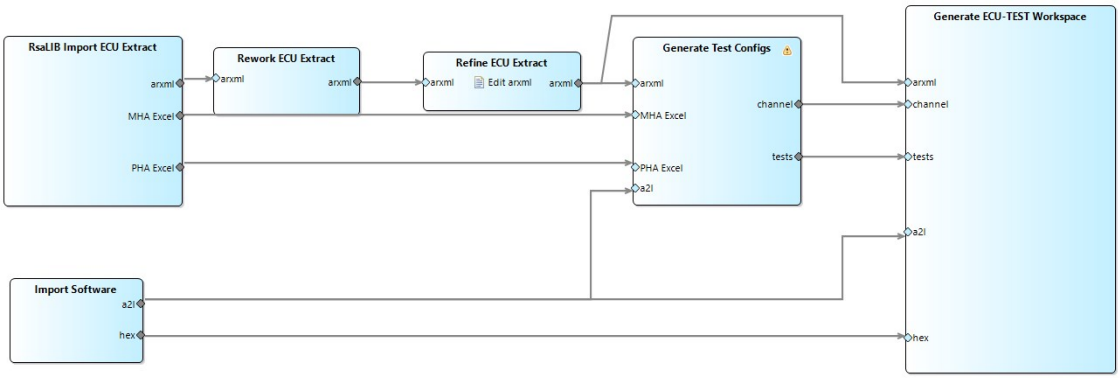

Figure 4.1. TestFlow Workflow used for this project

The development of the latter started from an already existing workflow from RSA workflow for classical CAN. The further modifications are operated on an xml file, edited on the editor provided by TestFlow as in Figure 4.2.

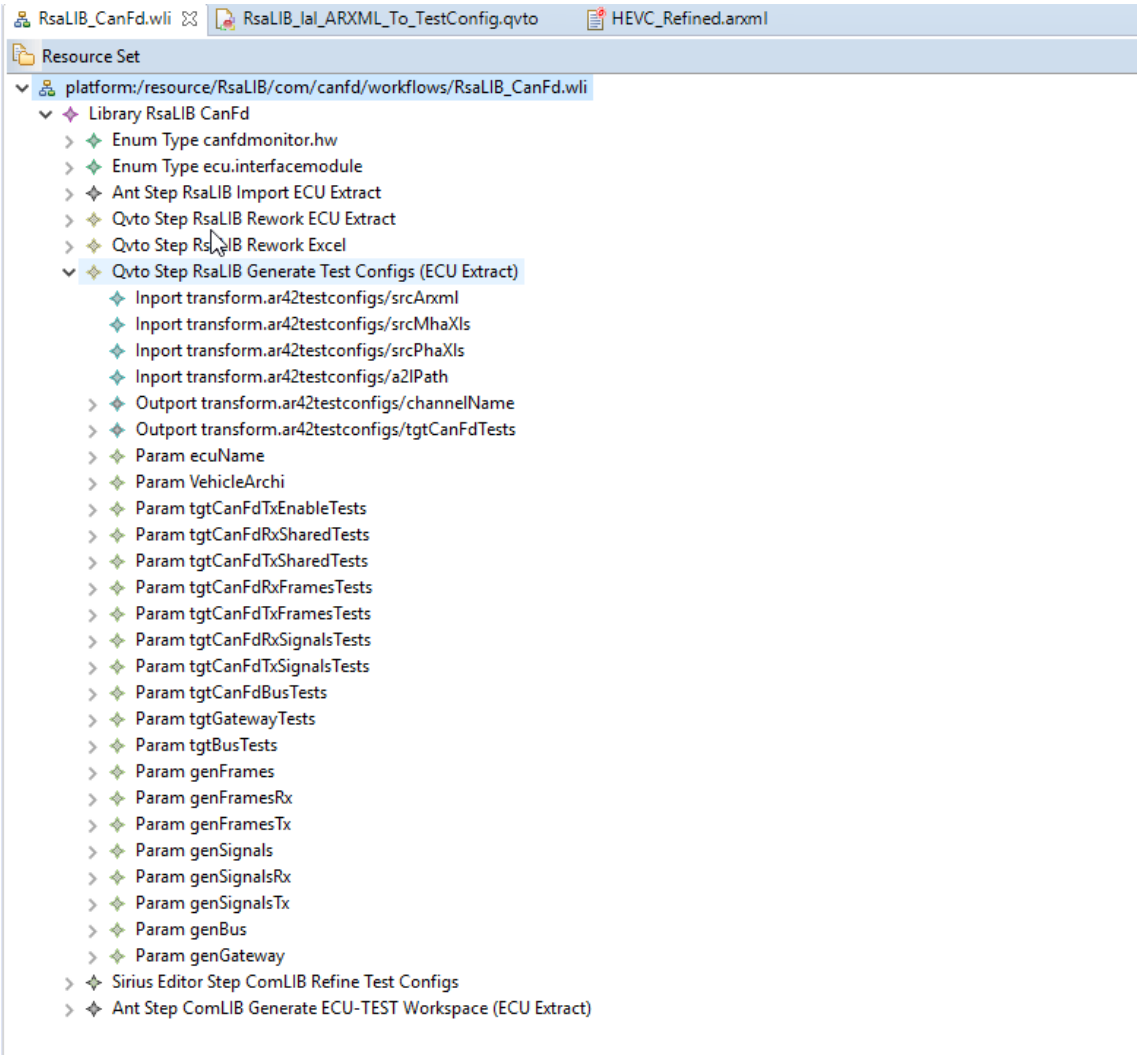

Figure 4.2. Workflow modifications on the editor

4.1.1 QVT-operational language

Each of the blocks belonging to the workflow corresponds to a qvto step. For instance, the block called "Generate test configs" in figure 4.1 corresponds to the "Qvto Step RsaLIB Generate Test Configs (ECU Extract)" in figure 4.2.

These scripts are written in **QVTO, QVT-Operational** language. QVT, which stands for "*Query-View-Transfer*", is a standard set of languages for model transformation defined by the Object Management Group. All of them operate on models which conform to Meta-Object Facility (MOF) 2.0 metamodels; the transformation states which metamodels are used. The QVT standard integrates the OCL 2.0 standard and also extends it with imperative features.

In particular, QVTO is an imperative language designed for writing unidirectional transformations.

4.1.2 Extraction of information

"RsaLIB import ECU Extract", “Import Software” and “Rework/Refine ECU Extract” extract the description and the specifications files and they refine them. The complete description of the input files can be found at 2.3.
The work was not focused on those, as their functionalities were very similar to those of the previous workflow.

4.1.3 Generation of test configurations

On the other hand, a relevant effort was given on the “Generate Test Configs” block: as we can see in figure 4.2 in the "Qvto Step RsaLIB Generate Test Configs" has several inputs:

- The "*Import transform.ar42testconfigs*" files which are some of the description files, previosly described in 2.3,
- Some "*Param*" such as the name of ecu, the boolean values to enable the generation of some part of the test, and so on and so forth.

Also, there are listed some outputs: the name of the channel, lately used in the generation of the test workspace, as well as **the .test files**.

This script takes each PDU (from a Regular CAN-FD Frame or a container frame) and it extracts the signals and the parameters that the test package needs as an input from the specification files.
Therefore, for every kind of test, this block automatically generates a file.test, where for each PDU to be set to this test, every parameter and every signal to map is listed.
Moreover, each of these mappings needs to follow a list of mapping objects, described in another script of the same format, in the library to which the test package belongs. In figure 4.3, the correspondences between the library file.test and the generated file.test are shown.

4.1.4 Generation of the workspace

The last block, “Generate ECU-TEST workspace” is the one that finally generates the ECU-TEST workspace.
The parameters for this block are:

- the interface module used to communicate with the ECU (→ Ethernet System)
- the calibration protocol (→ XETK)
- the kind of hardware support used for monitoring the bus (→ ES582)

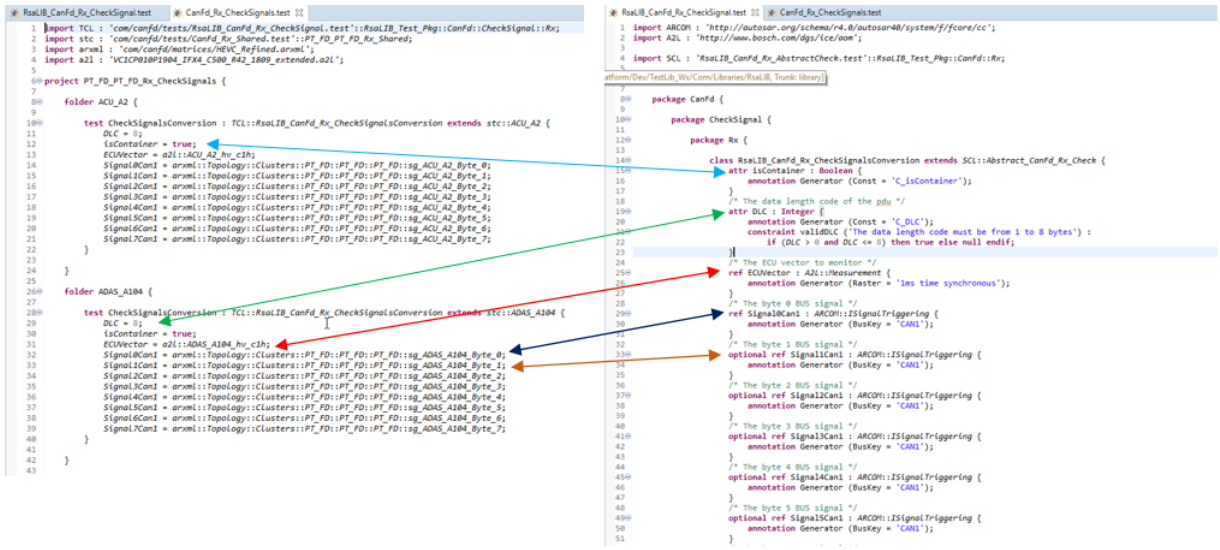

Figure 4.3. Correspondence between the .test from the library and the generated .test

- what kind of test bench is used (→ during the development, SOTB)

Referring to figure 4.4 and figure 2.7, we can see the correspondences between the hardware configurations and the parameters for the generation of the workspace.

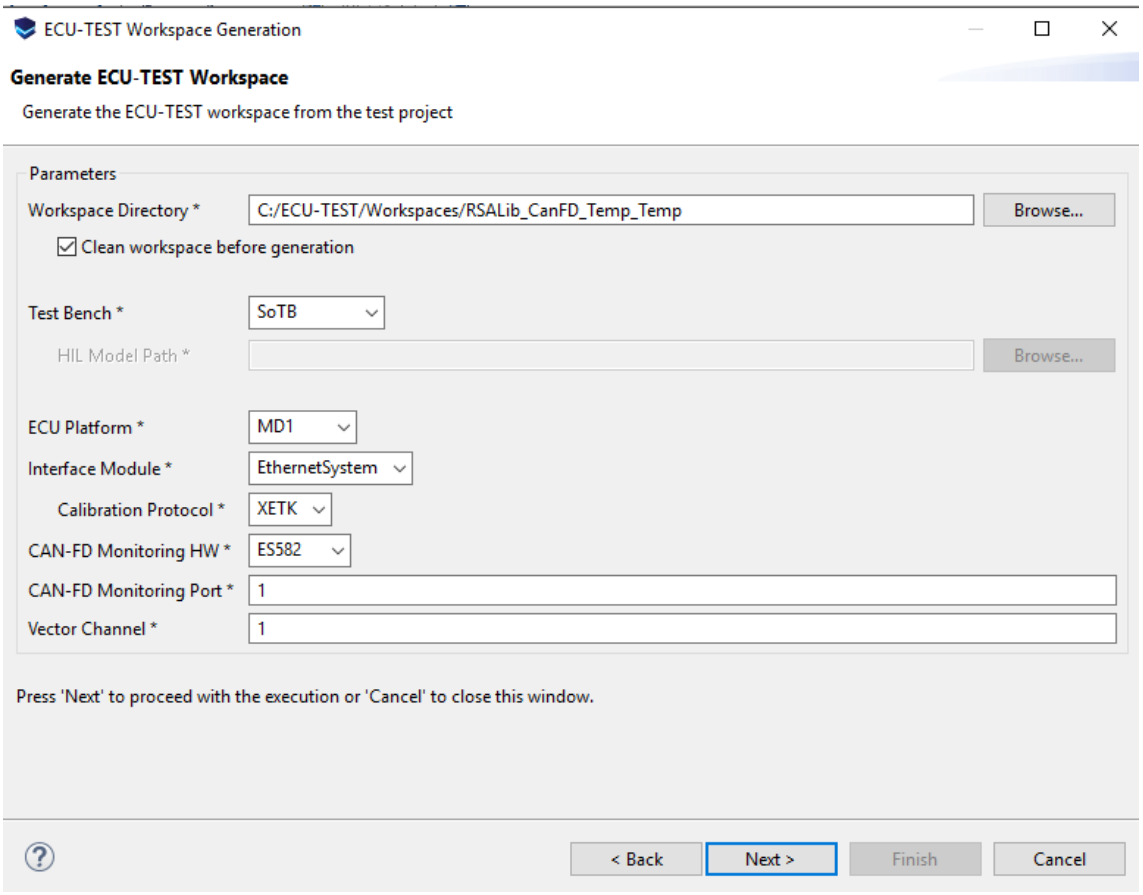

Figure 4.4. Generate ECU-TEST workspace step, configuration

Chapter 5

STEP 3

One of the most interesting features introduced in CAN-FD is the encrypted authentication system that protects the communication. This authentication system is based on:

- **MAC, message authenticaton code**, described in 5.1.3,
- **ARC, anti-replay code**, described in 5.1.1.

5.1 MAC/ARC

As shown in figure 2.4 and explained in section 2.1.1, the layout of the secured CAN data is as follow:

[Authentic_CAN DATA] [Padding] [LSB_ARC] [MAC]

5.1.1 ARC

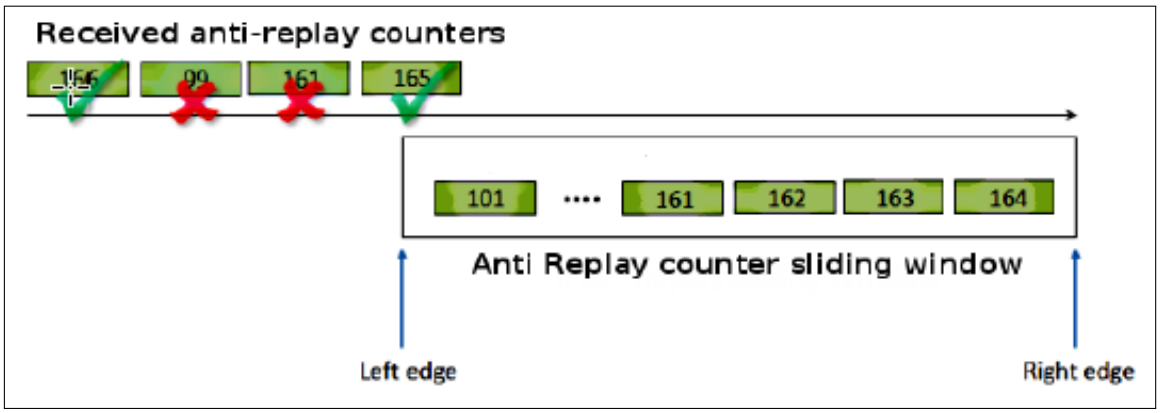

Figure 5.1. ARC

The anti-replay counter is a *monotonous counter saved in each ECU (6 bytes long).* The receiving ECU keeps track of which CAN frames it has already processed based on these Anti-Replay Counters with the use of a sliding window of all acceptable sequence numbers. The default

anti-replay window is 64 CAN frames wide, as shown in figure 5.1.
The possible scenarios when a CAN Frame is received are:

- If the ARC falls *within the window and was not previously received*, then the ECU verifies MAC of the received frame. If it is valid, the packet is accepted, and **marked as received**.
- If the ARC falls *within the window* and *was previously received*, the packet is **dropped**
- If the received Anti-Replay Counter is *greater than the highest sequence number* in the window, then the ECU verifies the MAC of the received frame. If it is valid, the packet is accepted, and **marked as received**. The sliding window is then moved to the right. The frame is also marked as received.
- If received Anti-Replay Counter is *less than the lowest sequence* in the window, the packet is **dropped**, and the corresponding error counter is incremented.

5.1.2 ARC Synchronization

The Anti-Replay Counters of the different emitting ECUs must be kept synchronizedbetween them. All secured ECUs shall write all current counter values to non-volatile memory when shutting down (go to sleep).
The following scenarios may force a de-synchronization (mismatch of ARC):

1. ECUs are not powered on at the same time
2. The network may delay or drop the transmission of CAN frames
3. ECUs may reset unexpectedly if power is lost for a sufficient period of time
4. Part replacement

For point 1 and 2 of the list above, the emitter continues to send secured CAN frames while the receivers do not receive the corresponding CAN frames.
If the number of missed CAN frames is less than 2^{16} (length of LSB_ARC in bits), then the receivers are able to recover by themselves.
If the number is $> 2^{16}$, then the full counter has to be transferred to the receivers.
In order to achieve this, all emitter ECUs will send their full counter in a low rate CAN frame called the synchronization frame. The time period of this synchronization CAN frame must be lower than the time required to increase the counter by 2^{16}. The **synchronization frame** includes a MAC.
For point 3 and 4, if the receivers reset and lose the current counter, then it will be re-synchronized with the low rate synchronization CAN frame sent by the emitter. If the emitter reset and lose the current counter, then the receivers will drop all secured CAN frames. The emitter must receive

the full counter from one (or more) receivers. When receiving more than X CAN frames that fail to authenticate (wrong MAC), receiver ECUs must send all the full anti-replay counters in a secured CAN frame called the **re-synchronization frame**. The re-synchronization frame includes a MAC.
In this work the full anti-replay counter will be named **FAR**.

5.1.3 MAC

The MAC value is the AES-CMAC output as defined in RFC4493 truncated to the most significant 64 bits (8 bytes). This algorithm has 3 inputs:

1. The message M
2. The secret AES key K
3. The length of the message in bytes len.

The output of the MAC generation algorithm is a 128-bit string, called a MAC, which is used to validate the input message.

AES-CMAC

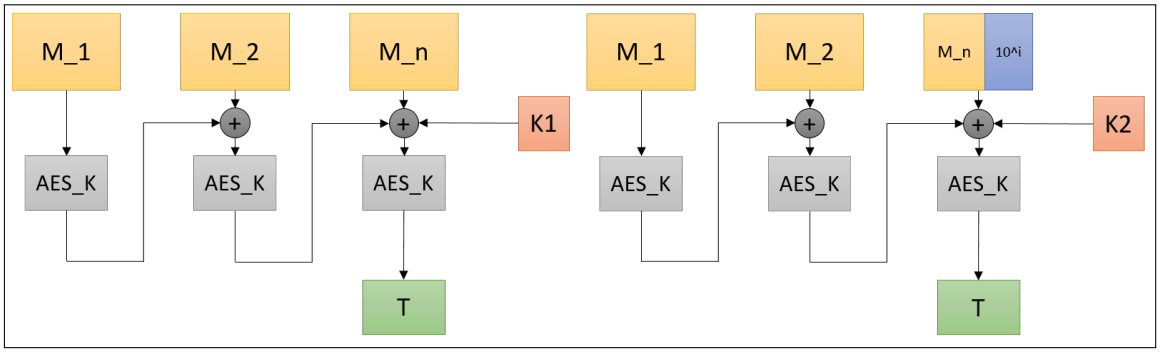

Figure 5.2. fig:AES-CMAC algorithm. Case1 on the right, Case 2 on the left.

The algorithm of AES-CMAC [1] starts with the generation fo the two subkeys K1 and K2, by the generate_subkeys algorithm.
In order to decide which of the subkeys the message M will be encrypted with, the length of the message M is analysed. Indeed, the message M is denoted by the sequence of M_i, where M_i is the i^{th} message block. That is, if M consists of n blocks, then M is written as:

$$M = M_1||M_2||...||M_{n-1}||M_n$$

The length of M_i is 128 bits for i = 1,...,n-1, and the length of the last block M_n is less than or equal to 128 bits (16 bytes).

There is a special treatment if the length of the message is not a positive multiple of the block length (128 bits):

- If M_n is 16 bytes long, nothing is done.
- If M_n is **less than 16 bytes long**, then it is padded with the bit-string 10^k, with k = $n - 16 * 8$ to adjust the length of the last block up to the block length.

The chain showed in figure 5.2 can start: M_1 is used as input of AES-K algorithm and its output will be bitewise xor-ed with M_2, and so on and so forth.
The chain continues until M_n. Here, if the length of the message M falls in the case 1, then K1 will be used to xor bitewise-lly M_n. Otherwise, K2 subkey will be used.
The MAC is the output of the last xor-ed block processed by AES-K algorithm.

AES-K algorithm

AES-CMAC uses the Advanced Encryption Standard AES-K [21] as a building block. AES-K is a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits.
It was established by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES is a subset of the Rijndael block cipher developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, who submitted a proposal to NIST during the AES selection process. Rijndael, as the algorithm is often called, is a family of ciphers with different key and block sizes. This algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256". For MAC/ARC in CAN-FD the size of the key adopted is of 128 bits (16 bytes). Hence, AES-128 is used.
The implementation of the above mentioned algorithm is not the purpose of this work. For further details, the reader can refer to [21].

5.1.4 Verifying MAC

When a secured frame is received, both MAC and ARC have to be verified. MAC verification is developed in these steps:

1. Extract tMAC from CAN frame by selecting the last 8 bytes.
2. Extract the LSBytes of the Anti-Replay Counter.
3. Extract the Authentic CAN_DATA by removing the Padding, LSB_ARC and tMAC from the secured frame.
4. Compose the message M to be encrypted.[1]

[1]The content of the message M cannot be specified in this dissertation for industrial secrecy.

5. Compute the AES-CMAC of message K, message M and length len.
6. Compare the result to the received MAC to be verified. If they are equal, then the message is valid, otherwise, it is invalid.

5.2 MAC/ARC: test development

The encrypted authentication system, described in 5.1 and present in the communication protocol, was one of the feature to test during this project. Since CAN-FD is the first communication protocol with this feature, a brand new test plan had to be implemented.
The ECU-TEST package developed is called "RsaLIB_CanFd_Tx_CheckMAC.pkg". Its aim is to test the MAC/ARC of the frames sent by the ECU under test.
This package is meant to be run for each of the Secured Frame sent by the ECU. For sake of simplicity, in this section we will name

- the tested secured frame as **SecFrame**
- the synchronization frame as **SyncFrame**.

In the stimulation part, the ECU is waken up by a "Wakeup frame" and check for the enable signal of SecFrame and SyncFrame. The recording of the frames sent on the channel is then started.
At this point, the test waits for an amount of time necessary to have at least one SyncFrame transmission from the ECU under test. As described in 5.1.2, the range of time should be about $2^{16} * P_{\text{SynchFrame}}$.
Also, the test has to wait for a significant number of SecFrames to be sent. The range of time to be waited is computed as $maxP_{\text{SecFrame}} * 5,2000ms$. Next, the recording is stopped and the analysis part is started.

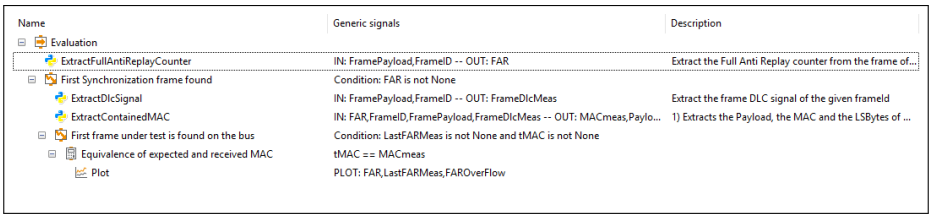

Figure 5.3. CheckMAC trace step analysis

In the trace step analysis, shown in figure 5.3, the .asc recording of all frames on the CAN-FD bus is analised. Here, the analysis can start from the time stamp in which the first SynchFrame is found. This is done because the FAR is essential to calculate the message M[2].

[2]The content of the message M cannot be specified in this dissertation for industrial secrecy.

Here, the first FAR extraction script is launched, written by the student. The script is called "ComLIB_CanFd_ExtractFullAntiReplayCounter".[3]
Once FAR has been correctly extracted, another script is launched, developed by the student and called "ComLIB_CanFd_ExtractContainedMAC". The tasks covered by the latter are:

1. it extracts the Payload, the MAC and the LSBytes of ARC of the SecFrame,
2. it composes the message M to be encrypted,
3. it generates the expected MAC.
4. it compares the expected MAC to the one received in the Payload.

In particular, the point 3 is operated by a third python script developed by the student, called "MAC.py". Its inputs are the secret AES key K, the FAR, the SecFrame data field and its ID. The script can be found in the appendix 7.6.
Weather at point 4 the two MAC do not coincide, a failure is launched.

5.2.1 FAR synchronization

During the run of this test package, there was the possibility that the SynchFrame was sent after the overflow of the 2 LSBytes read in the SecFrame. An example of recording is shown here below 5.2.1:

timestamp1 CANFD 1 SecFrameID 00 04 5E FF 39
... other frames...
timestamp2 CANFD 1 SecFrameID 00 04 5F 00 55

In the example, we can see that the second SecFrame is sent *after* the overflow of the first 2 LSBytes occurred, causing an erroneous MAC calculation in the trace step analysis.
For this, two solutions have been analysed. The first solution proposed was of **counting all the secured frames** sent and received on the CAN-FD bus and implement a real internal anti-replay counter inside the test environment. Nevertheless, this solution was discarded because of the necessity of including every secured frame of the architecture inside the mapping of the test package, which would have considerably load the generation of the workspace and of the test run.
The second solution was modifying "ComLIB_CanFd_ExtractContainedMAC", described in 5.2: when comparing the AES-CMAC of SecFrame to the received MAC to be verified, if the first try is not successfull (i.e. if the two MACs are not equivalent), the MAC is calculated again considering the FAR as overflow-affected. At this point the internal FAR value is incremented of 0x1000 and the MAC.py script is called once again. The comparison is done again:

[3]The content of this script cannot be furtherly described due to industrial secrecy.

- if the two MAC values are equivalent, a log message is sent in order to report the FAR overflow,
- otherwise, a failure is launched.

The code of this modification of the "ComLIB_CanFd_ExtractContainedMAC" is shown in the appendix, at 7.7.

5.3 ARXML File: solving the dependencies

As described in the section 2.3.1, the addition of the description of a feature was essential to correctly access it and validate it.
Given that the changes on this file could not be operated immediately, alternative solutions had to be found in order to develop the test and make them bug-free.
The first example provided in 2.3.1 - mapping signals on bytes – was faced by temporarily modifying the .arxml, changing the mapping of 4 frames, two in transmission and two in reception, one of kind Container and one of Regular per type, in order to have a set of correctly described PDUs with whom it was possible to run the tests. The student implemented this operation.
For the second issue - generating and reading the MAC/ARC part – it was important to consider also that the tool ECU-TEST *cannot manage to generate nor to receive this security part of the frame.*
This gave as a result the necessity of implementing in the analysis of the received secured frames a function that could skip the recognition of the MAC/ARC byte in the payload, whether the frame was secured.
Firstly, it was important to pass to functions the parameter of MACLen and ARCLen. Seen that these parameters were still not available in the .arxml file and that – as said in 5.1 – the parameter was already known for this project RsaLIB frames, it was decided to force these parameter in the mapping: Instead of: Furthermore, for the analysis of received secured frames it was used

```
if(self.isFrameSecured()){
    //logWarning("Secured Pdu: "+containerPdu.shortName);
    var secureContainerPdu : ARCOM::SecuredIPdu = containerPdu.map getSecuredPdu();
    SecuredParams->put('MACLen',8*8);
    SecuredParams->put('ARCLen',2*8);
}
```

Figure 5.4. Forcing MACLen and ARCLen

the function extractSecuredByteStream, which cuts the payload in payload, MAC and ARC. For example, for this frame:

00 04 5F 04 C5 24 FF 28 00 00 14 26 6D 84 94 2D EE A8 9E FD

```
var securePdu : ARCOM::SecuredIPdu = self.iPdu.map getSecuredPdu();
securedParams->put('MACLen',securePdu.authenticationProps.authInfoTxLength);
securedParams->put('ARCLen',securePdu.freshnessProps.freshnessValueTxLength);
```

Figure 5.5. Original extraction of MACLen and ARCLen

The result of the function is:
Frame: XXX_AYYYSC_FD 'MAC': ByteStream('6D:84:94:2D:EE:A8:9E:FD'), 'ARC': ByteStream('14:26'), 'Payload': ByteStream('00:04:5F:04:C5:24:FF:28:00:00').
Subsequently, the analysis is performed only on payload.
Considering that this same class was used to perform analyses on the transmitted frames, it was important that also these needed a security part to be sent. Seen that it was still immature to send a correct MAC and a correct ARC, the first approach was to at least instantiate the bytes when transmitting the frames to the ECU: With this method, a frame with this format could be

```
if(self.isSecured):
    paddSize = self.CalculatePaddingSize()
    byteStreams[idx] = byteStream+ByteStream('00')*paddSize+ByteStream(range(1,self.MACLen+self.ARCLen+1))
else :
    byteStreams[idx] = byteStream
```

Figure 5.6. Sending a dummy vector for a secured frame

correctly sent to the ECU:

00 00 20 08 FF FF FF FF FF FF FF FF 00 00 01 02 03 04 05 06 07 08 09 0A

Nevertheless, the tool, before of starting the trace analysis, analyzed this frame. The result was that the tool, which blocked the execution of the trace analysis, raised this error:
"Problem at timestamp 0.598063 (in file): Corrupt PDU header detected: id=329223, dlc=8 remaining bytes=2" This error made us realize that ECU-TEST automatically recognized other PDUs:

1. FF FF FF FF FF FF FF FF, with header 00 00 20 and dlc = 8
2. 03 04 with header 00 00 01 and dlc = 2
3. 09 0a with header 05 06 07 (329233 in decimal, as in the launched error)and dlc = 8

The latter PDU raised the error because of the incoherence between the declared DLC, 8, and the number of databytes left, 2.
The temporary solution was to place only 0s in the MAC/ARC dummy vector, in order not to give any fake DLC to the tool analysis.
Moreover, before adding the MAC/ARC bytes, it was necessary to fill the missing bytes of the fixed frame length with the PaddingBytes (see 2.1.1).

CalculatePaddingSize is the function that the student developed to calculate the number of padding bytes. This calculi is based on how many of its PDUs are present in the sent frame, what size is their header and what is frame's MACLen and ARCLen.
The code for the function is in section 7.5 of the appendix. As we can see from figure 5.6, CalculatePaddingSize is called for each sent PDU belonging to a secured frame. Since for this set of test, it was possible to take for granted that only one PDU per frame could be sent per time.
This piece of code in the appendix in 7.5 was added to the already mentioned Frame.py, in the class Calculate Frame.

Chapter 6

Bonus Step: Gateway

Gateway definition A gateway is "a central hub that securely and reliably interconnects and transfers data across the many different networks found in vehicles. It provides physical isolation and protocol translation to route signals between functional domains (powertrain, chassis and safety, body control, infotainment, telematics, ADAS) that share data" [3].
The gateway controller plays a fundamental role as providing secure, seamless communications between networks and ECUs, including bridging between the many internal networks of the vehicle, such as CAN (low, high speed), LIN, FlexRay, and Ethernet protocols and the external networks of the outside world.
The smooth transfer of data is essential for ensuring ECUs have the information they need for proper vehicle operation, so the gateway must provide **any-to-any network** communications and with **low latency and jitter**.

6.1 Gateway in ComStack

In the ComStack (see 1.8.1), this feature can be shown in figure 6.1. There exist some frames that are "bridged": once they are sent on the *bus under test*, the ComStack has to resend it on another bus, called *destination bus*. For sake of simplicity, in this section the bus under test will be called **ch1** and the destination bus will be named as **ch2**.
The first requirement to be covered was to **verify the presence of the bridged frame on the ch2, after being sent on the ch1**. The bridged frame must contain the **same value**.
In order to develop the package, it was necessary to correctly read and record the traces on the two buses involved, which will be called ch1 and ch2. By saying "correctly" we mean that the measurement have to be *synchronous* and on the *same time base*. Using the tool ECU-TEST, two different ways were provided to do the dual recording:

1. CANalyzer: the recording is completely operated by the tool Canalyzer, which reads the two buses at the same time. Then, it collects all the frames recorded in the time range in a unique .asc file, where the bus name is specified for each of them (In the test folder, Verification on

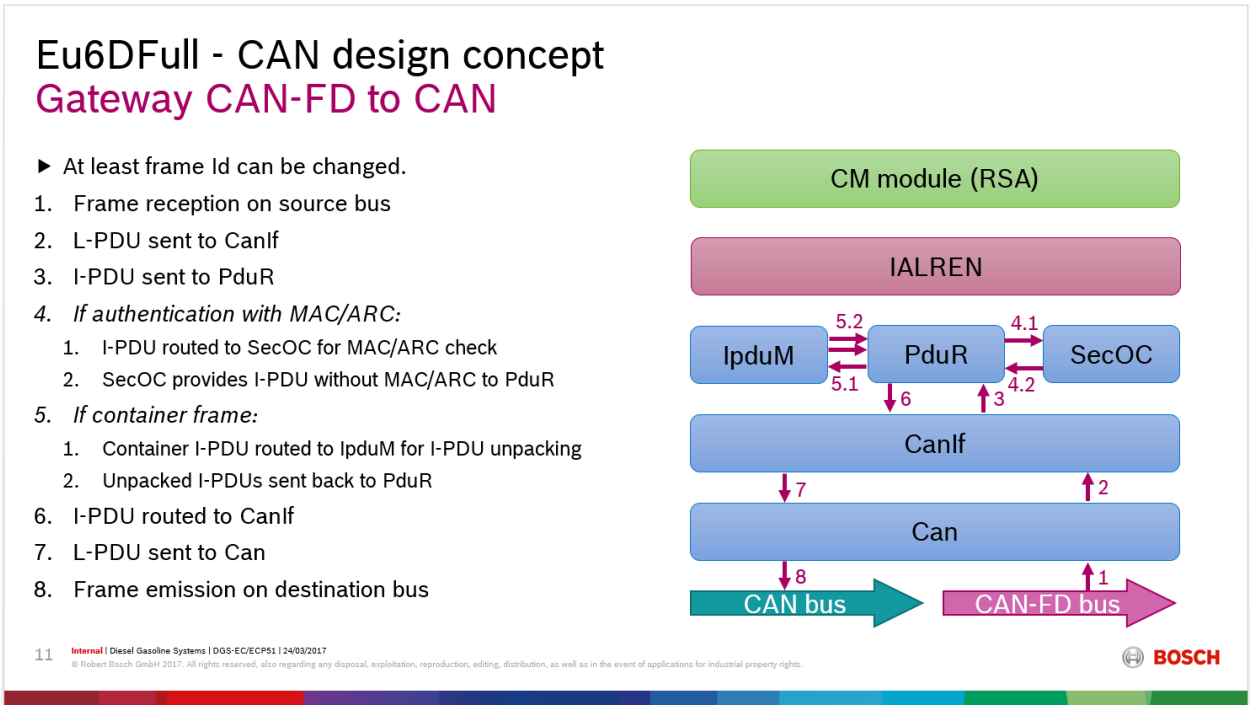

Figure 6.1. Gateway in ComStack

the recordings↪logging.asc/logging_2.asc).

2. VectorHW: the two recordings of the two channels are separately operated by ECU-TEST . The frames present on ch1 are recorded in Recording.asc and those on ch2 in Recording_2.asc.

6.1.1 Verification of the recordings

The idea behind the choice between the two methods or recording lies on the question: can the two separate recording of VectorHW be as efficient and correct as the unique one operated by canalyzer? In order to verify this, the Package "VerificationOnTheRecordings.pkg" was developed: ch1 is stimulated with a frame belonging to the bridged collection. The frame is supposed to change at any emission. The 3 recording are launched: CANalyzer, VHW_ch1 and VH_ch2. Then the recording files (.asc) are analyzed by the mean of a python script ComLIB_CanFd_CheckRecordings (trace step template):

- Its inputs are the frame payload of ch1 and ch2, and the frameid of ch1 and ch2
- A necessary parameter is the value of the ID of the frame under test.
- As soon as the script finds an occurrence of the frame under test in the trace of ch1, it looks for the corresponding occurrence on ch2.

- The elapsed time between the 2 events is measured and stored.

This script was repeated twice, once for the CANalyzer recording and on the 2 VHW recordings.

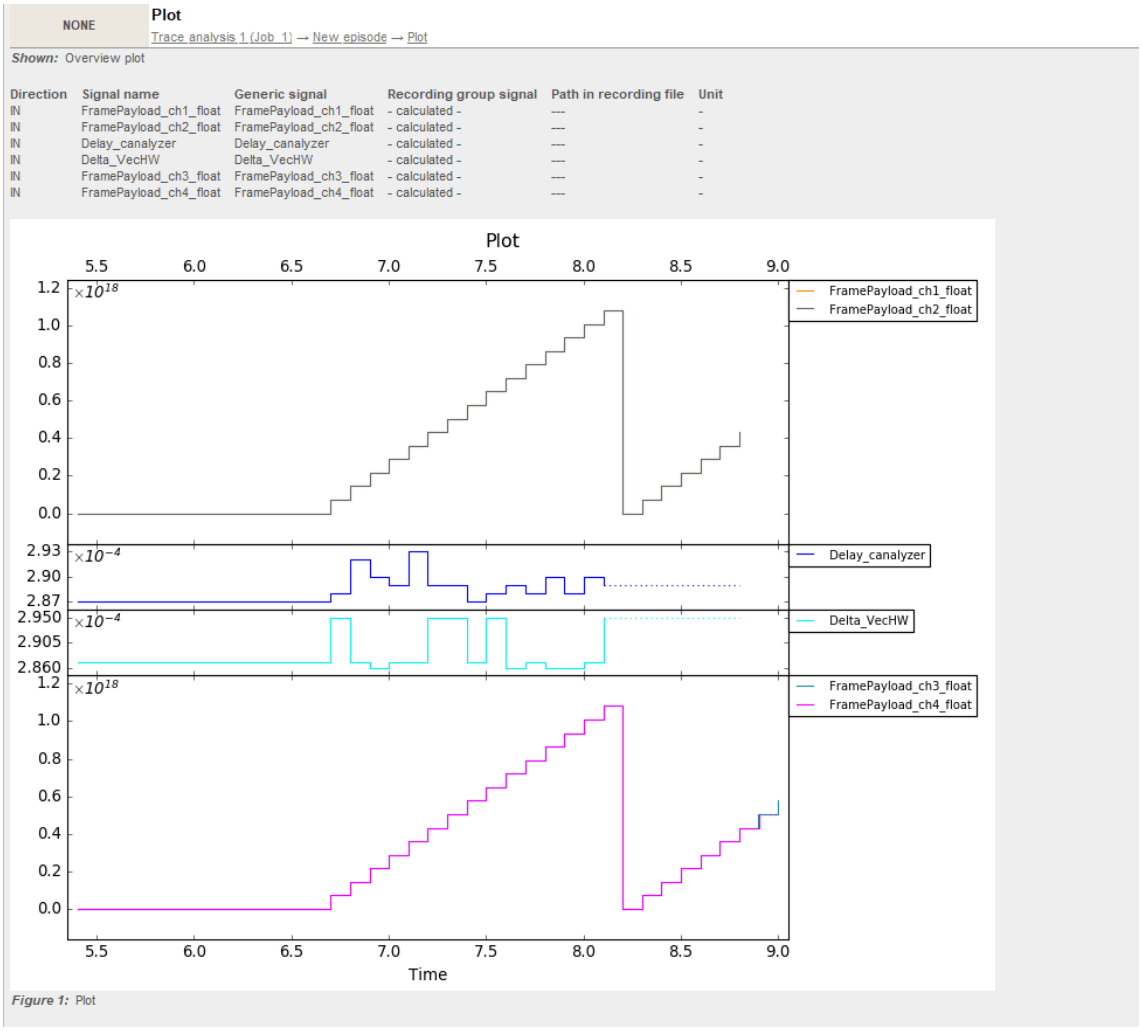

Figure 6.2. Plotting the bridged frame

The collections of elapsed time between the same events were then compared by the mean of a plot shown in figure 6.2.
The idea behind this package was that the Canalyzer recording was for sure more reliable, as it is done by another tool, whose scheduling may be less influenced by the load of the ECU-TEST tool. The hypothesis is to have a certain dT between 2 events on ch1 and ch2. These two events will be recorded on both canalyzer and VectorHW. The willing is to prove that the two dT of the two recordings are the same or with a tolerable difference. If this is proven, we could use the VectorHW

recording instead of the Canalyzer one, way more complicate to automatically configure and start. The results are those in figure 6.3. It results that the two dT, Delta_canalyzer and Delta_VectHW

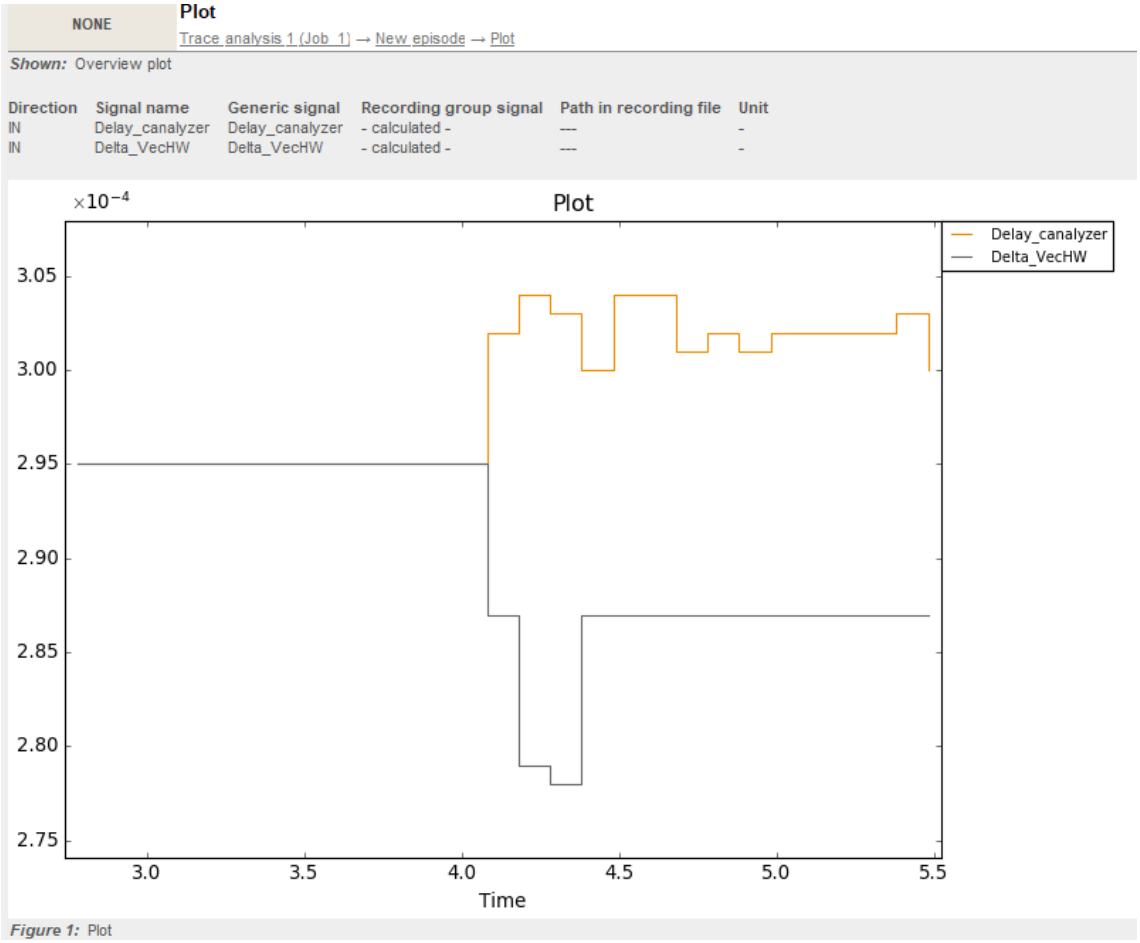

Figure 6.3. Plotting delays of a bridged frame

are never the same but, generally speaking, DeltaVecHW tends to be lower of 10ms. Since the data were somewhat difficult to interpret, that the time of my internship was running out and finally that the main aim was to roughly verify the presence of the bridged message and not its timing, it was decided to go on with the project.
Hence, it was decided to put off the issue of better analyzing the difference between the two kinds of recording and decide to which of those apply the python script (trace step template).

6.1.2 Bridged frame presence: test development

Following the analysis described in 6.1.1, it was possible to develop a test package and a python script for processing the recordings of the two signals, independently from the source of the two

recording.
The test package was called "RsaLIB_CanFd_Rx_CheckGWMessagePresence" and its tasks are:

1. The test package starts the recordings of the two channels;
2. Then, it stimulates the ch1 bus with 16 different frames. Each emission must be different from its previous; indeed the passages for the stimulation were taken by the package checkpduscheduling.
3. The recordings are stopped.
4. Then, in the trace analysis the new trace step template(ComLIB_CanFd_Gateway) is applied to the ch1 and ch2 recordings:
   - Each frame with the ID of bridged frame has to be found first on the ch1, then on ch2. If one of those frames is missed, or if the order is not respected, a failure is launched.
   - Once the corresponding occurrence is found on ch2, the elapsed time is measured.

6.1.3 Bridged frame presence: test automation

Once the package described in 6.1.2 was completed, it was necessary to generate the project in the ECU-TEST workspace.

As usual, this was done by modifying the qvto script RsaLIB_Ial_ARXML_To_TestConfig.

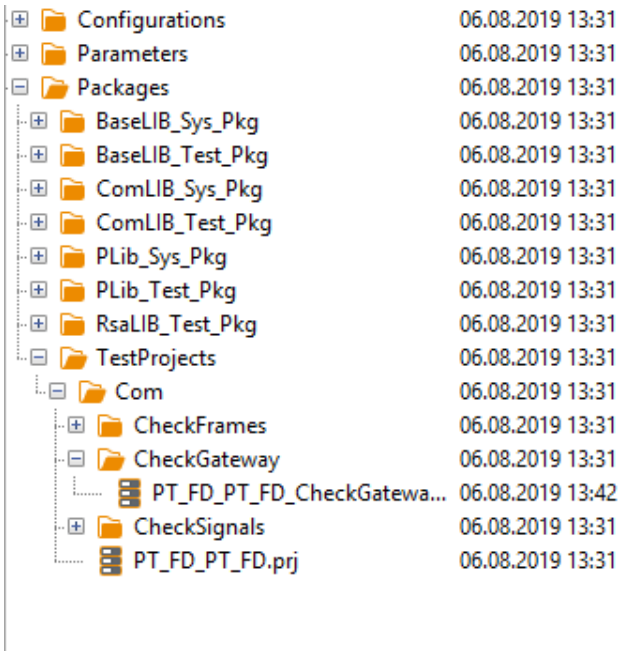

Figure 6.4. Automation test

The result in the Explorer of ECU-TEST is shown in figure 6.4.

Chapter 7

Progress and conclusions

7.1 Progress

Firstly, the training took the very first three weeks of the experience, and it was fundamental to achieve the comprehension of the network, the interfaces, the physical tools and the software tools.
Afterwards, for STEP1 (see chapter 3), a particular attention was given to the requirements of the client, RSA group, and, based on the latter, the planning of this internship.
Every 3 weeks, the student had to present the state of progress to the internship manager and to the responsible for ComLIB tests and custom libraries.
By the end of April, the test packages developed covered all the requirements of the CAN-FD test. In addition, whether a package tests different specifications for different parameters, two PDUs, one container and one non-container, were tested for each property.
As an example, RsaLIB_CanFd_Rx_CheckFrameScheduling, which tests the PDUs with and without MessageCounter property.
Moreover, for each of the new package or python script, full tests were presented, in order to guarantee the ability of the new piece of code of detecting erroneous behavior and calibration.
It was not always straightforward to find a solution for simulating erroneous behavior of the signals. For instance, for the case of adaptation of "ComLIB_CheckIncrementalSignal" for the requirement Rx : Non-MessageCounter, since the arrangement of the calculator and the channel could not get to the point of accelerating the frames at the point of the ECUbuffer to miss some of them, decision was taken to analyze a fake signal record, in .csv format.
Some of the delay during this path was also due to the different procedures to have the licenses for installing software, access to internet and other servers.
By mid-May, also STEP2 (chapter 4) was completed, which involved the generation of the ECU-TEST workspace by the mean of TestFlow. After the first generation, minor issues were found in the launching of the complete test, like the missing security encryption of the transmitted frames and the synchronization of the various signal recordings presents in the packages.
By the end of May, all the tests are correctly mapped and correctly working.

It was possible to find a bug in the reception of a certain kind of frames, when their PDUs are sent with a DLC greater by the declared one: despite the message was correctly present on the bus, their ECU buffer were not updated. This detection was reported to the client.
For what concerns STEP3, see chapter 5, the testing of the whole set of secured frames sent by the ECU under test is completed.
A complete Gantt diagram is shown in appendix, at 7.8.

7.2 Future work

As explained in 2.3.1, due to the limitations of the ECU-TEST tool, it was not possibile to complete the test package for the secured frames sent to the ECU under test. Nevertheless, the implementation of this test package can lay on the scripts developed by the student, in particular on MAC.py (see 7.6), which can generate a correct MAC.
The last issue to solve is the ARC generation, on which Mr Sarrazin is currently working on.
For the gateway step, a first draft of test package was developed to cover the requirement of presence of the bridged frame on the destination channel. However, in the project only the frames bridged between the channel under test and another channel are tested.
The future work involves:

1. a deeper analysis on the correctness of the parallel recording of VectorHW tool - see 6.1.1-;
2. the refinement of this package;
3. the refinement of the automation of test generation.

Morever, once point 1 is completed, another package to cover the timing requirements on the bridged frames shall be developed.
Finally, the full test plan shall be launched in the HIL environment.

7.3 Impressions

In the first weeks, I have seen how much the tools that are used in the academic field -less powerful and on a smaller scale- and those used in the company can be different. As an example, in the first days, towards my training, I was often showed the tests running on LABcar, an HIL testbench. It was common that my tutor and I agreed on how such a huge and powerful tool inevitably leads to recurring crashes.
This was the main reason why my testbench was not LABCar, as well as in this site only few are present and a fast development of the tests was required.
Moreover, I found interesting to see applied the agile methodologies to plan the activities of the team and the way to present the coverage of objectives: to me, before this experience, they were concepts seen in university courses, never applied to my work.

In addition, an important role during the project was played by the integration in the team and the acquaintanceship with the other members of other teams, especially whom it was important to cooperate with. The innate internationality of the company ensured that the environment is multicultural and this helped my colleagues and me to interact easily, in both English and French. I found a warm welcoming from the team and this lightened the stress of the new experience. Nevertheless, I cannot deny that speaking a little of French was essential.

7.4 Environmental and societal impact

7.4.1 Environmental impact

The environmental impact is an important issue nowadays and I could not miss the opportunity of giving my contribution.
Firstly, choosing a little test bench - with a low voltage implant of 12V - was more environmental friendly than a huge HIL test bench.
Besides, the project on which I am currently working is for an ECU in a hybrid car, which is the new tendency of the automotive industry that finally meets the need of a carbon free environment.
In addition, as presented in the section 1.7, Bosch is opening to new technologies as automated driving, electric and hybrid vehicles, which are a great solution for the issue of pollution and the need of low ecological impact.
Furthermore, Bosch launched a new campaign for carbon neutrality by 2020: the goal is to make the emissions carbon neutral worldwide in the company's direct sphere of influence.
This is done by sourcing "green" electricity from existing renewable energy generation facilities and all energy efficiency in general, as stricter shutdown management of both electricity and heating.
Moreover, by 2030, Bosch wants to generate 400 GWh of its energy needs from own renewable resources and to save 1.7 TWh of energy – equivalent to the annual electricity consumption of all private households in a city like Cologne.

7.4.2 Automated testing: societal impact

Software testing is one of the critical part in software development, which ensures quality of a system and represents the ultimate review of specification, design, and coding.
An interesting new branch of this discipline is **Automated Testing**, on which this project is based. This is the method of controlling the execution of tests and, lately, comparing actual test results with predicted or expected results, by making use of special software tools. The automated execution of the test plans involves a previous phase of *test scripts development* using scripting languages such as Python, JavaScript or TCL, so that test cases can be executed by computers with **minimal human intervention** and attention.
In literature, automated testing is often opposed to the approach of **Manual Testing**, for which the test engineer prepares test cases and manually executes them to identify defects in the software.

Here, the tester plays an important role of manually launching every single test case, analysing the output logs and producing the final documentation.
The main differences shown between the two approaches in literature are:

1. Manual Testing tends to be more **time consuming and tedious**, harmful aspect for a company, where the competencies of each employee need to be exploited minute by minute and where time-to-market is a main issue.

2. As a consequence of point 1, Manual Testing is more prone to induct **human error** and it is **less reliable** [20].

3. Automated Testing inherits some qualities of the software, like the **repeatability** by the test tool, the **reusability** on different versions of the software and the **suitability to regression**, as we could see in 3, during which I adapted the already existing test plan for CAN to the CAN-FD protocol.

4. Seen the point 3, Automated Testing results to be more **cost effective**, as the test cases are run by the automation tool so less testers are required.

Indeed, automated testing was introduced to overcome the overhead and dependencies on the human error of manual testing.
Nevertheless, during this project, I could find that my work in testing automation was likely to be time consuming as well. As an example, I remarked that parametrizing a test plan for each of the frame could be very complicate. Some of the test plans developed in step1 resulted in errors or failures at the end of step2, due to peculiarities of certains frames that I could not know before or I did not take into account.
Also the automation tool can put significant limitations to the parametrizing process. As an example, during the adaptation of the test package RsaLIB_CanFd_Rx_CheckFrameScheduling, it was necessary to put a limit on the PDU contained in a Frame, as in the trace analysis of ECU-TEST, it was not possible to map a variable number of signals.
In addition, automating the test of big systems - as an entire communication equipment of a group of ECU - has its overhead too, as it has to lay on modelling and various software and hardware tools, which the employee needs to properly master.
Moreover, skills require training, therefore time and money spent on it, as for instance the three weeks that I needed to read and understand the documentation, to approach to the tools and to install my test bench.
As a matter of fact, going through the litterature on automated and manual testing, one can find plenty of papers that analyse the effectiveness and the return of investments of the two methods. Many of these propose mathematical models to compute the cost model of both manual and automated testing. One of these is [16], which comes up with a first mathematical model, that

takes into account parameters as the expenditure for test specification and implementation and the expenditure for single test execution.
On the other hand, more recent papers like [19] differentiate the concept of cost of test *execution* for manual testing and test case *development* in automated testing. In addition, other parameters as fixed budget are taken into account, as well as other influence factors as the maintenance costs for automated tests, the early/late return on investment and the higher initial effort required by automated testing [15].
Nevertheless, most of the papers regarding the cost model analysis and the reporting of the experience of migration from manual to automated testing agree on the fact that "automated testing needs a higher initial effort, mainly caused by the creation of the scripts, but this cost can be amortized in time" [15].
However, what is the compromise to balance the higher investment?
First, the more the development is expensive, the more has to be reusable, flexible to changes and different calibrations, in order to be executed multiple times for regression testing.
Second, it is important to develop general and custom implementations, in order to re-use the software, which can be suitable to our case, just like the case of ComLIB and RsaLIB (see section 2.2.3).
Third, to properly document and describe the software in order to make it understandable from the next employees that could need to use it as I was asked when presenting my work.
Finally, to store the software on appropriate platform, accessible to the company employee – Bosch uses SVN servers – or to everyone, i.e. on GitHub.

Glossary

ECUbuffer For every signal present in a PDU, a Buffer is present in the ECU for storing the signal to be sent or received.. 32, 34–36, 42–44, 46, 71

functional safety It is the part of the overall safety of a system or piece of equipment that depends on automatic protection operating correctly in response to its inputs or failure in a predictable manner (fail-safe). The automatic protection system should be designed to properly handle likely human errors, hardware failures and operational/environmental stress.. 19

symmetric a type of encryption scheme in which the same key is used both to encrypt and decrypt messages.. 58

TCL Tool Command Language. 73

TestFlow Internal tool to generate automatically a test workspace, adapting to each part of the system under test the proper tests and calibrations. 25, 34, 49, 71

Acronyms

DLC Data Length Code (DLC), which expresses the length of the next message format. In CAN-FD we have FrameDLC and PDUDLC. 18, 28, 38, 40, 41, 43, 62, 72

MAC/ARC it is a short piece of information used to authenticate a message in other words, to confirm that the message came from the stated sender and it has not been changed. All messages have the same security level, so the selected design relies on a unique secret MAC key distributed among all producers and consumers of these messages. The MAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers to detect any changes to the message content. Anti-Replay Counter, it is a counter that both the sender and the receiver of a frame keep updated whenever the latter is sent or received. It is part of the security part in order to prevent replay attacks.. 19, 25, 29, 34, 41, 42, 59, 61, 62

MOF The Meta-Object Facility (MOF) is an Object Management Group (OMG) standard for model-driven engineering.. 50

PDU Primary Data Unit. It is a specific block of information transferred over a network. It is often used in reference to the OSI model, since it describes the different types of data that are transferred from each layer. 23, 24, 28–30, 34, 37–41, 46

RSA Renault S.A. is a French automotive company which manufactures, sells and distributes passenger cars, light commercial vehicles and associated components under the Renault brand (global), the Dacia brand (Europe and North Africa) and the Renault Samsung Motors brand (South Korea). 24, 33, 35, 37, 38, 40, 49, 71

Appendices

7.5 CalculatePaddingSize

```
def CalculatePaddingSize (self):
        #We take for granted that only one PDU is sent and
        #that the header size is 'Short',
        #that MACLen = 2 Bytes and ARCLen = 8 Bytes
        PaddingSize = 0
        tempSize = self.hSize + self.MACLen + self.ARCLen+self.dlc
        #log.WPrint("tempSize"+str(tempSize)+" hSize: "+str(self.hSize)+" dlc: "+
            ↪ str(self.dlc) + " MACLen: "+ str(self.MACLen)+ " ARCLen: "+str(self
            ↪ .ARCLen))
        if (tempSize > 0 and tempSize <= 8):
            PaddingSize = 0
        elif (tempSize > 8 and tempSize <= 12):
            PaddingSize = 12 - tempSize
        elif (tempSize > 12 and tempSize <= 16):
            PaddingSize = 16 - tempSize
        elif (tempSize > 16 and tempSize <= 20):
            PaddingSize = 20 - tempSize
        elif (tempSize > 20 and tempSize <= 24):
            PaddingSize = 24 - tempSize
        elif (tempSize > 24 and tempSize <= 32):
            PaddingSize = 32 - tempSize
        elif (tempSize > 32 and tempSize <= 48):
            PaddingSize = 48 - tempSize
        elif (tempSize > 48 and tempSize <= 64):
            PaddingSize = 48 - tempSize
        else :
            log.EPrint("Error occured during Padding Size Calculation. Header Size
                ↪  + MACLen + ARCLen > 64 byte: " + str(tempSize) )
            raise

        if (PaddingSize < 0):
            log.EPrint("Error occured during Padding Size Calculation. Padding
                ↪ Size is smaller than 0: " + str(PaddingSize) )
            raise
        else :
            return PaddingSize
```

7.6 MAC.py

```
import log
from lib.common.dataElements.ByteStream import ByteStream
import math

class SecuredObj:
    def __init__(self, key):
        self._key=[]
        for i in range(0, 16*2, 2):
            self._key.append(int(key[i:i+2],16))
        self._block = [0]*16
        self._encBlock = [0]*16
        self._expandedKey = [0]*176
        self._e = [0]*4
        self._encryptState = False
        self._roundKey = [0]*16
        self._ecolumn0 = [0]*4
        self._ecolumn1 = [0]*4
        self._ecolumn2 = [0]*4
        self._ecolumn3 = [0]*4
        self._constRb = [0]*15+[0x87]

    def VerifyMAC(self, param):
        self._ARC = param['FAR']
        self._Payload = param['Datafield']
        self._frameID = param['frameID']

        self.__init_param()

        tMAC = self.aes_cmac(self._key, self._message.Ints(), len(self._message.
            ↪ Ints()))
        return tMAC

    def __init_param(self):
        self._message = ############################
        //This code line was obscured due to industrial secrecy.
        self._block = [0]*16
        self._encBlock = [0]*16
        self._expandedKey = [0]*176
        self._e = [0]*4
        self._encryptState = False
        self._roundKey = [0]*16
```

```
        self._ecolumn0 = [0]*4
        self._ecolumn1 = [0]*4
        self._ecolumn2 = [0]*4
        self._ecolumn3 = [0]*4
        self._constRb = [0]*15+[0x87]

    def aes_cmac (self, key, input, length):
        K1=[0]*16
        K2=[0]*16
        X=[0]*16
        Y=[0]*16
        M_last=[0]*16
        padded=[0]*16
        n = (length+15)/16
        #log.WPrint("n= "+str(n)+"length = "+str(length))
        self.generateSubKey(key, K1, K2)
        #log.WPrint('Key1: '+str(ByteStream(K1))+' , Key2: '+str(ByteStream(K2)))
        if n == 0:
            n = 1
            flag = 0
        else:
            if (length%16) == 0:
                flag = 1
            else:
                flag = 0
        if flag == 1:
            self.xor128(input[16*(n-1):], K1, M_last)
            #log.WPrint("M_last XOR K1: "+str(ByteStream(M_last)))
        else:
            padded = self.padding(input[16*(n-1):], length%16)
            self.xor128(padded, K2, M_last)
        for i in range(n-1):
            #log.WPrint('debug5. Processing bytes from '+str(16*i)+', until '+str
                ↪ (16(i+1)))
            self.xor128(X, input[16*i:16*(i+1)], Y)

            #log.WPrint('xor nel for di aesmac. X = '+str(X)+'input['+str(16*i)
                ↪ +':'+str(16*(i+1))+']='+str(input[16*i:16*(i+1)])+', Y='+str(Y)
                ↪ )
            X = self.encrypt(Y, key, 16)
        ##log.WPrint("for end")
        self.xor128(X, M_last, Y)
```

```
        #log.WPrint("last xor")

        X = self.encrypt(Y, key, 16)
        ##log.WPrint("last encrypt")
        #log.WPrint("X: "+str(X))
        return X

    def padding(self, lastb, length):
        pad = [0]*16
        for j in range(16):
            if j<length:
                pad[j] = lastb[j]
            elif j==length :
                pad[j] = 0x80
            else:
                pad[j] = 0x00
        return pad

    #generatesubkeys fatta con i piedi!
    def generateSubKey(self, key, K1, K2):
        L = [0]*16
        Z = [0]*16
        tmp = [0]*16
        L=self.encrypt(Z, key, 16)
        if (L[0]&0x80)==0:
            #log.WPrint('(L[0]&0x80)==0. L= '+str(ByteStream(L)))
            self.lefshift_onebit(L, K1)
        else:
            self.lefshift_onebit(L, tmp)
            self.xor128(tmp, self._constRb, K1)
        if (K1[0]&0x80)==0 :
            log.WPrint('(K1[0]&0x80)==0. K1= '+ hex(K1[0]))
            self.lefshift_onebit(K1, K2)
        else:
            self.lefshift_onebit(K1, tmp)
            self.xor128(tmp, self._constRb, K2)

    def xor128(self, a, b, out):
        for i in range(16):
            out[i] = (a[i] ^ b[i])%0x100
    def lefshift_onebit(self, input, output):
        overflow = 0
```

```
        for i in range(15,-1,-1):
            #log.WPrint('overflow? leftshift, for, i='+str(i))
            output[i] = (input[i] <<1)&0xFF
            #log.WPrint("output["+str(i)+"] = "+str(output[i])+" input["+str(i)+"]
                ↪  = "+str(input[i]))
            output[i] = (output[i] | overflow)&0xFF
            #log.WPrint("output["+str(i)+"] = "+str(output[i])+" overflow = "+str(
                ↪ overflow))
            if (input[i]&0x80) :
                overflow=1
            else:
                overflow = 0
    def encrypt(self, input, secretKey, inSize):
        nbrRounds = 10
        encOut = [0]*16
        expandedKeySize = 16 * (nbrRounds + 1)
        for g in range(4):
            for j in range(4):
                self._encBlock[g+(j*4)] = input[(g*4)+j]
        #log.WPrint('encblock in encrypt : '+str(self._encBlock))
        self.expandKey(secretKey, inSize, expandedKeySize )
        self.aes_Main(nbrRounds)
        for k in range(4):
            for l in range(4):
                encOut[(k*4)+l] = self._encBlock[k+(l*4)]
        return encOut

    def aes_Main(self, num_rounds):
        self.createRoundKey(self._encryptState)
        self.addRoundKey(self._encryptState)
        self.createRoundKey(self._encryptState)
        i=1
        while i < num_rounds:
            #log.WPrint("Round: "+str(i))
            self.createRoundKey(16*i)
            self.aes_Round(self._encryptState)

            i = i + 1
        #log.WPrint("Round: 10")
        self.subBytes(False)
        self.shiftRows(False)
        self.createRoundKey(16*num_rounds)
```

```
        self.addRoundKey(16*num_rounds)

        #log.WPrint("end of aes_main")

    def createRoundKey(self, roundKeyPointer):
        '''if roundKey is False:
            roundKey = 0'''
        #log.WPrint("round key = "+str(roundKeyPointer))
        for i in range(4):
            for j in range (4):
                self._roundKey[(j*4)+i] = self._expandedKey[roundKeyPointer+(i*4)+
                    ↪ j]
        #log.WPrint("createRoundKey: "+str(ByteStream(self._roundKey)))

    def addRoundKey(self, isInv):
        for i in range(16):
            if isInv is True:
                self._block[i] = (self._block[i]^self._roundKey[i])%0x100
            else:
                #log.WPrint('ADDRoundKey: encblock in encrypt at position ['+str(i
                    ↪ )+']: '+str(self._encBlock[i])+'roudkey at position ['+str(i
                    ↪ )+']: '+str(self._roundKey[i]))
                self._encBlock[i] = (self._encBlock[i]^self._roundKey[i])%0x100


    def aes_Round(self, isInv):
        if isInv is not True:
            self.subBytes(False)
            #log.WPrint("substitution: "+str(ByteStream(self._encBlock)))
            self.shiftRows(False)
            #log.WPrint("shiftRows "+str(ByteStream(self._encBlock)))
            self.mixColumns(False)
            #log.WPrint("MixColumn "+str(ByteStream(self._encBlock)))
            self.addRoundKey(False)
            #log.WPrint("AddRoundKey "+str(ByteStream(self._encBlock)))
        else:
            self.shiftRows(isInv)
            self.subBytes(isInv)
            self.addRoundKey(isInv)
            self.mixColumns(isInv)
        #log.WPrint("end of aes_Round")
```

```
def subBytes(self, isInv):
    for i in range(16):
        if isInv is True:
            self._block[i]=self.getSBoxInvert(self._block[i])
        else:
            #log.WPrint('SUBBYtes: self._encBlock['+str(i)+']='+str(self.
                ↪ _encBlock[i]))
            self._encBlock[i] = self.getSBoxvalue(self._encBlock[i])
    #log.WPrint("end of subbytes")

def shiftRows(self,isInv):
    ##log.WPrint('debug: shiftRows start')
    for i in range(4):
        self.shiftRow(i*4,i,isInv)
    ##log.WPrint('debug: shiftRows start')

def mixColumns(self, isInv):
    ##log.WPrint('debug: mixCoulumns start')
    d = 0
    if isInv is not True:
        for i in range(16):
            mod = i%4
            if mod ==0:
                self._ecolumn0[d] = self._encBlock[i]
                #break
            elif mod == 1:
                self._ecolumn1[d] = self._encBlock[i]
                #break
            elif mod == 2:
                self._ecolumn2[d] = self._encBlock[i]
                #break
            elif mod == 3:
                self._ecolumn3[d] = self._encBlock[i]
                d = d + 1
    else:
        for i in range(16):
            mod = i%4
            if mod ==0:
                self._ecolumn0[d] = self._block[i]
                #break
            elif mod == 1:
                self._ecolumn1[d] = self._block[i]
```

```
                #break
            elif mod == 2:
                self._ecolumn2[d] = self._block[i]
                #break
            elif mod == 3:
                self._ecolumn3[d] = self._block[i]
                d = d + 1
                #break
            '''else:
                break'''
    ##log.WPrint('debug: mixCoulumns end')
    self.mixColumn(False)


def mixColumn(self, isInv):
    gl = self.polynomialMul
    mult = [0]*4
    temp0 = [0]*4
    temp1 = [0]*4
    temp2 = [0]*4
    temp3 = [0]*4
    if isInv is not True:
        mult[0] = 2
        mult[1] = 1
        mult[2] = 1
        mult[3] = 3
        for i in range(4):
            temp0[i] = self._ecolumn0[i]
            temp1[i] = self._ecolumn1[i]
            temp2[i] = self._ecolumn2[i]
            temp3[i] = self._ecolumn3[i]
        self._ecolumn0[0] = (gl(temp0[0], mult[0]) ^ gl(temp0[3], mult[1]) ^
            ↪ gl(temp0[2], mult[2]) ^ gl(temp0[1], mult[3]))
        self._ecolumn1[0] = (gl(temp1[0], mult[0]) ^ gl(temp1[3], mult[1]) ^
            ↪ gl(temp1[2], mult[2]) ^ gl(temp1[1], mult[3]))
        self._ecolumn2[0] = (gl(temp2[0], mult[0]) ^ gl(temp2[3], mult[1]) ^
            ↪ gl(temp2[2], mult[2]) ^ gl(temp2[1], mult[3]))
        self._ecolumn3[0] = (gl(temp3[0], mult[0]) ^ gl(temp3[3], mult[1]) ^
            ↪ gl(temp3[2], mult[2]) ^ gl(temp3[1], mult[3]))
        self._encBlock[0] = self._ecolumn0[0]
        self._encBlock[1] = self._ecolumn1[0]
        self._encBlock[2] = self._ecolumn2[0]
```

```
        self._encBlock[3] = self._ecolumn3[0]

        self._ecolumn0[1] = (gl(temp0[1], mult[0]) ^ gl(temp0[0], mult[1]) ^
            ↪ gl(temp0[3], mult[2]) ^ gl(temp0[2], mult[3]))
        self._ecolumn1[1] = (gl(temp1[1], mult[0]) ^ gl(temp1[0], mult[1]) ^
            ↪ gl(temp1[3], mult[2]) ^ gl(temp1[2], mult[3]))
        self._ecolumn2[1] = (gl(temp2[1], mult[0]) ^ gl(temp2[0], mult[1]) ^
            ↪ gl(temp2[3], mult[2]) ^ gl(temp2[2], mult[3]))
        self._ecolumn3[1] = (gl(temp3[1], mult[0]) ^ gl(temp3[0], mult[1]) ^
            ↪ gl(temp3[3], mult[2]) ^ gl(temp3[2], mult[3]))
        self._encBlock[4] = self._ecolumn0[1]
        self._encBlock[5] = self._ecolumn1[1]
        self._encBlock[6] = self._ecolumn2[1]
        self._encBlock[7] = self._ecolumn3[1]

        self._ecolumn0[2] = (gl(temp0[2], mult[0]) ^ gl(temp0[1], mult[1]) ^
            ↪ gl(temp0[0], mult[2]) ^ gl(temp0[3], mult[3]))
        self._ecolumn1[2] = (gl(temp1[2], mult[0]) ^ gl(temp1[1], mult[1]) ^
            ↪ gl(temp1[0], mult[2]) ^ gl(temp1[3], mult[3]))
        self._ecolumn2[2] = (gl(temp2[2], mult[0]) ^ gl(temp2[1], mult[1]) ^
            ↪ gl(temp2[0], mult[2]) ^ gl(temp2[3], mult[3]))
        self._ecolumn3[2] = (gl(temp3[2], mult[0]) ^ gl(temp3[1], mult[1]) ^
            ↪ gl(temp3[0], mult[2]) ^ gl(temp3[3], mult[3]))
        self._encBlock[8] = self._ecolumn0[2]
        self._encBlock[9] = self._ecolumn1[2]
        self._encBlock[10] = self._ecolumn2[2]
        self._encBlock[11] = self._ecolumn3[2]

        self._ecolumn0[3] = (gl(temp0[3], mult[0]) ^ gl(temp0[2], mult[1]) ^
            ↪ gl(temp0[1], mult[2]) ^ gl(temp0[0], mult[3]))
        self._ecolumn1[3] = (gl(temp1[3], mult[0]) ^ gl(temp1[2], mult[1]) ^
            ↪ gl(temp1[1], mult[2]) ^ gl(temp1[0], mult[3]))
        self._ecolumn2[3] = (gl(temp2[3], mult[0]) ^ gl(temp2[2], mult[1]) ^
            ↪ gl(temp2[1], mult[2]) ^ gl(temp2[0], mult[3]))
        self._ecolumn3[3] = (gl(temp3[3], mult[0]) ^ gl(temp3[2], mult[1]) ^
            ↪ gl(temp3[1], mult[2]) ^ gl(temp3[0], mult[3]))
        self._encBlock[12] = self._ecolumn0[3]
        self._encBlock[13] = self._ecolumn1[3]
        self._encBlock[14] = self._ecolumn2[3]
        self._encBlock[15] = self._ecolumn3[3]
    else:
```

```
mult[0] = 0xe
mult[1] = 0xb
mult[2] = 0xd
mult[3] = 0x9
for i in range(4):
    temp0[i] = self._ecolumn0[i]
    temp1[i] = self._ecolumn1[i]
    temp2[i] = self._ecolumn2[i]
    temp3[i] = self._ecolumn3[i]

self._ecolumn0[0] = (gl(temp0[0], mult[0]) ^ gl(temp0[1], mult[1]) ^
    ↪ gl(temp0[2], mult[2]) ^ gl(temp0[3], mult[3]))
self._ecolumn1[0] = (gl(temp1[0], mult[0]) ^ gl(temp1[1], mult[1]) ^
    ↪ gl(temp1[2], mult[2]) ^ gl(temp1[3], mult[3]))
self._ecolumn2[0] = (gl(temp2[0], mult[0]) ^ gl(temp2[1], mult[1]) ^
    ↪ gl(temp2[2], mult[2]) ^ gl(temp2[3], mult[3]))
self._ecolumn3[0] = (gl(temp3[0], mult[0]) ^ gl(temp3[1], mult[1]) ^
    ↪ gl(temp3[2], mult[2]) ^ gl(temp3[3], mult[3]))
self._block[0] = self._ecolumn0[0]
self._block[1] = self._ecolumn1[0]
self._block[2] = self._ecolumn2[0]
self._block[3] = self._ecolumn3[0]

self._ecolumn0[1] = (gl(temp0[0], mult[3]) ^ gl(temp0[1], mult[0]) ^
    ↪ gl(temp0[2], mult[1]) ^ gl(temp0[3], mult[2]))
self._ecolumn1[1] = (gl(temp1[0], mult[3]) ^ gl(temp1[1], mult[0]) ^
    ↪ gl(temp1[2], mult[1]) ^ gl(temp1[3], mult[2]))
self._ecolumn2[1] = (gl(temp2[0], mult[3]) ^ gl(temp2[1], mult[0]) ^
    ↪ gl(temp2[2], mult[1]) ^ gl(temp2[3], mult[2]))
self._ecolumn3[1] = (gl(temp3[0], mult[3]) ^ gl(temp3[1], mult[0]) ^
    ↪ gl(temp3[2], mult[1]) ^ gl(temp3[3], mult[2]))
self._block[4] = self._ecolumn0[1]
self._block[5] = self._ecolumn1[1]
self._block[6] = self._ecolumn2[1]
self._block[7] = self._ecolumn3[1]

self._ecolumn0[2] = (gl(temp0[0], mult[2]) ^ gl(temp0[1], mult[3]) ^
    ↪ gl(temp0[2], mult[0]) ^ gl(temp0[3], mult[1]))
self._ecolumn1[2] = (gl(temp1[0], mult[2]) ^ gl(temp1[1], mult[3]) ^
    ↪ gl(temp1[2], mult[0]) ^ gl(temp1[3], mult[1]))
self._ecolumn2[2] = (gl(temp2[0], mult[2]) ^ gl(temp2[1], mult[3]) ^
    ↪ gl(temp2[2], mult[0]) ^ gl(temp2[3], mult[1]))
```

```
        self._ecolumn3[2] = (gl(temp3[0], mult[2]) ^ gl(temp3[1], mult[3]) ^
            ↪ gl(temp3[2], mult[0]) ^ gl(temp3[3], mult[1]))
        self._block[8] = self._ecolumn0[2]
        self._block[9] = self._ecolumn1[2]
        self._block[10] = self._ecolumn2[2]
        self._block[11] = self._ecolumn3[2]

        self._ecolumn0[3] = (gl(temp0[0], mult[1]) ^ gl(temp0[1], mult[2]) ^
            ↪ gl(temp0[2], mult[3]) ^ gl(temp0[3], mult[0]))
        self._ecolumn1[3] = (gl(temp1[0], mult[1]) ^ gl(temp1[1], mult[2]) ^
            ↪ gl(temp1[2], mult[3]) ^ gl(temp1[3], mult[0]))
        self._ecolumn2[3] = (gl(temp2[0], mult[1]) ^ gl(temp2[1], mult[2]) ^
            ↪ gl(temp2[2], mult[3]) ^ gl(temp2[3], mult[0]))
        self._ecolumn3[3] = (gl(temp3[0], mult[1]) ^ gl(temp3[1], mult[2]) ^
            ↪ gl(temp3[2], mult[3]) ^ gl(temp3[3], mult[0]))
        self._block[12] = self._ecolumn0[3]
        self._block[13] = self._ecolumn1[3]
        self._block[14] = self._ecolumn2[3]
        self._block[15] = self._ecolumn3[3]

def polynomialMul(self, a, b):
    """Galois multiplication of 8 bit characters a and b."""
    p = 0
    for counter in range(8):
        if b & 1: p ^= a
        hi_bit_set = a & 0x80
        a <<= 1
        # keep a 8 bit
        a &= 0xFF
        if hi_bit_set:
            a ^= 0x1b
        b >>= 1
    return p


def getSBoxInvert(self, value):
    rsbox = //huge matrix, not written
    ##log.WPrint('debug: getSBoxInvert end')
    return(rsbox[value])

def shiftRow(self, statePointer, nbr, isInv):
    ##log.WPrint('debug: shiftRow start')
```

```
        tmp=[0]*4
        for i in range(nbr):
            for j in range(4):
                if isInv is True:
                    tmp[j] = self._block[statePointer+j]
                else:
                    tmp[j] = self._encBlock[statePointer+j]
            if isInv is True:
                self._block[statePointer] = tmp[3]
                self._block[statePointer+1] = tmp[0]
                self._block[statePointer+2] = tmp[1]
                self._block[statePointer+3] = tmp[2]
            else:
                ##log.WPrint('SHIFTRO: before transformation encblock in encrypt :
                    ↪  '+str(self._encBlock))
                self._encBlock[statePointer] = tmp[1]
                self._encBlock[statePointer+1] = tmp[2]
                self._encBlock[statePointer+2] = tmp[3]
                self._encBlock[statePointer+3] = tmp[0]
                ##log.WPrint('SHIFTRO: after transformation encblock in encrypt :
                    ↪ '+str(self._encBlock))
        ##log.WPrint('debug: shiftRow end')

    def expandKey(self, key, inSize, expandedKeySize):
        rconIteration = 1
        currentSize = 0
        ##log.WPrint("entrata in expandedKey")
        for j in range(inSize):
            self._expandedKey[j] = key[j]
            ##log.WPrint("primo for di expanded key. j="+str(j))

        currentSize += inSize
        while currentSize < expandedKeySize :
            ##log.WPrint("nel while di expandedkey. currentSize="+str(currentSize)
                ↪ )
            temp = currentSize - 4


            self._e = self._expandedKey[currentSize-4:currentSize]

            if currentSize%inSize==0 :
                self.core(rconIteration)
```

```
            rconIteration += 1


        for m in range(4) :
            if currentSize>=expandedKeySize:
                break
            ##log.WPrint('e['+str(m)+']: '+str(self._e[m])+'expandedKey[
                ↪ currentSize-inSize] = '+str(self._expandedKey[currentSize-
                ↪ inSize])+', currentSize= '+str(currentSize))
            self._expandedKey[currentSize] = (self._expandedKey[currentSize-
                ↪ inSize]^self._e[m])%0x100
            currentSize += 1
    #log.WPrint('expanded key: '+str(self._expandedKey))

def core(self, iteration):
    self.rotate(self._e)
    for l in range(4):
        self._e[l] = self.getSBoxvalue(self._e[l])
        ##log.WPrint('e['+str(l)+']: '+str(self._e[l]))
    rcon_value = self.getRconValue(iteration)

    self._e[0] = (self._e[0]^rcon_value)%0x100

def rotate(self, enc):
    tmp=[0]*4
    for i in range(4):
        tmp[i] = enc[i]

    enc[0] = tmp[1]
    enc[1] = tmp[2]
    enc[2] = tmp[3]
    enc[3] = tmp[0]

def getRconValue(self, num):
    ##log.WPrint('getRconValue. num = '+str(num))
    Rcon = //huge matrix, not written.
    return(Rcon[num])

def getSBoxvalue(self, var):
    SBox = //huge matrix, not written
    #log.WPrint("In SBoxValue, var = "+str(var)+ ", SBox["+str(var)+"] = "+str
        ↪ (SBox[var]))
```

```
        return(SBox[var])
```

7.7 FAR overflow

```
#if the first attempt of MAC calculation does not succeed, another calculation is
    ↪ launched
          #with the FAR updated as if an overflow occurred.
          if (dataContainer.isValidMAC[:8]!=SecuredCollection['MAC']):
              FARnew += 0x10000
              FARByteStream = fromIntToByteStream(FARnew, 6)
              param = {'FAR': FARByteStream, 'frameID': IDByteStream, 'Datafield
                  ↪ ': SecuredCollection['Payload'][:DataLength] }
              dataContainer.isValidMAC = ByteStream(dataContainer.secObj.
                  ↪ VerifyMAC(param))

              #if the calculated MAC is now correct, we can say that the
                  ↪ overflow occurred
              if (dataContainer.isValidMAC[:8]==SecuredCollection['MAC']):
                  FAROverFlow = True
                  log.WPrint(&quot;Far overflow occurrence. Last FAR from synchro
                      ↪  frame: &quot;+str(fromIntToByteStream(FAR, 8))+&quot;,
                      ↪ FAR calculated: &quot;+str(FARByteStream))
```

7.8 Gantt Diagram

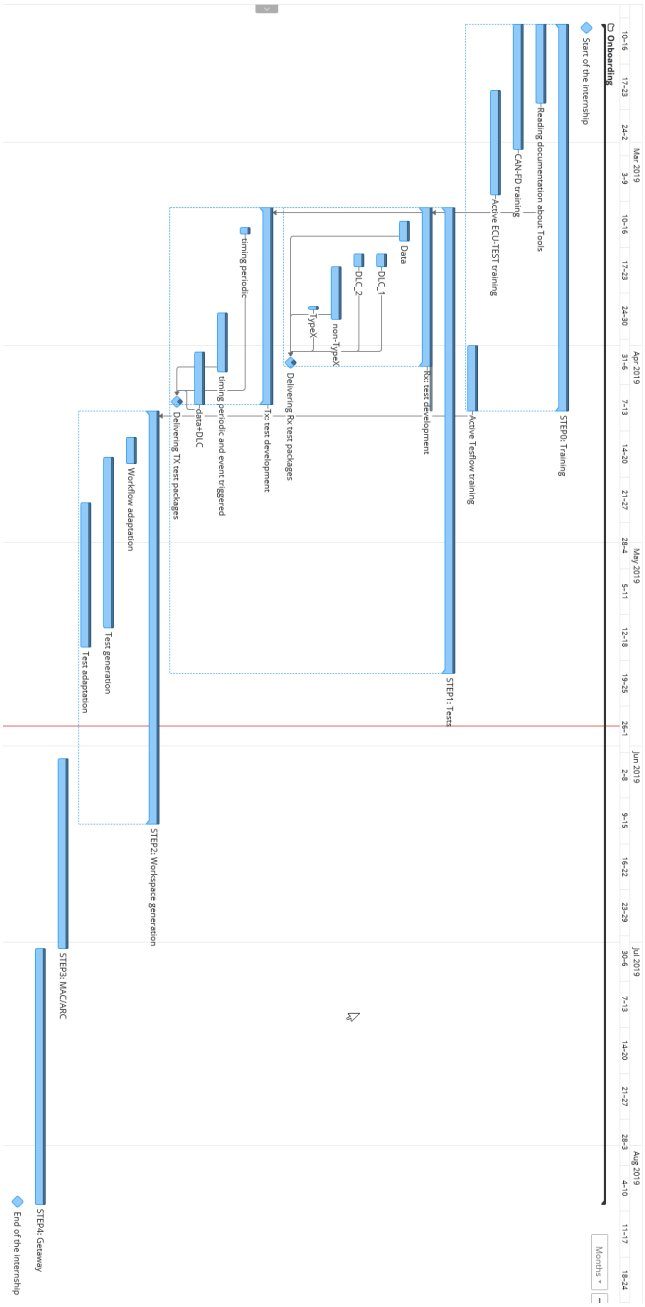

Figure 7.1. Gantt Diagram

Bibliography

[1] The aes-cmac algorithm. https://tools.ietf.org/html/rfc4493#section-2.4. Accessed: 2019-11-15.

[2] Automotive electronic control units (ecu) solutions. https://www.embitel.com/product-engineering-2/automotive/control-units. Accessed: 2019-10-18.

[3] Automotive gateway. https://www.st.com/en/applications/telematics-and-networking/automotive-gateway.html. Accessed: 2019-10-30.

[4] Automotive security in a can. https://www.electronicdesign.com/automotive/automotive-security-can. Accessed: 2019-11-15.

[5] Car software: 100m lines of code and counting. https://www.linkedin.com/pulse/20140626152045-3625632-car-software-100m-lines-of-code-and-counting/. Accessed: 2019-11-15.

[6] Codebases. https://informationisbeautiful.net/visualizations/million-lines-of-code/. Accessed: 2019-11-15.

[7] Ecu. https://www.magnetimarelli.com/business_areas/powertrain/gasoline-system-gdi/ecu. Accessed: 2019-10-18.

[8] Electronic control unit. https://en.wikipedia.org/wiki/Electronic_control_unit. Accessed: 2019-10-18.

[9] Hackers remotely kill a jeep on the highway—with me in it. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. Accessed: 2019-11-15.

[10] Hackers reveal nasty new car attacks–with me behind the wheel (video). https://www.forbes.com/sites/andygreenberg/2013/07/24/hackers-reveal-nasty-new-car-attacks-with-me-behind-the-wheel-video/#22de7420228c. Accessed: 2019-11-15.

[11] V-model (software development). https://en.wikipedia.org/wiki/V-Model_(software_development). Accessed: 2019-10-24.

[12] What is an electronic control unit? https://www.pistonheads.com/features/ph-features/what-is-an-electronic-control-unit-ph-explains/37771. Accessed: 2019-10-18.

[13] What's the difference between nrz, nrzi, and manchester encoding? https://www.electronicdesign.com/communications/what-s-difference-between-nrz-nrzi-and-manchester-encoding. Accessed: 2019-10-20.

[14] Robert Bosch et al. Can specification version 2.0. *Rober Bousch GmbH, Postfach*, 300240:72, 1991.

[15] Ignacio Dobles, Alexandra Martínez, and Christian Quesada-López. Comparing the effort and effectiveness of automated and manual tests. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2019.

[16] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[17] O Garnatz and P Decker. Can fd with dynamic multi-pdu-to-frame mapping. In *Proc. iCC 2015*, pages 05–8, 2015.

[18] Udo W Pooch, Denis Machuel, John McCahn, Udo W Pooch, Denis Machuel, John McCahn, Udo W Pooch, Denis Machuel, John McCahn, Udo W Pooch, et al. Osi-reference model-the is0 model of architecture for open system interconnection. In *Telecommunications and Networking*, volume 28, pages 409–450. Houghton Mifflin Englewood Cliffs, NJ, 1991.

[19] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91. ACM, 2006.

[20] RM Sharma. Quantitative analysis of automation and manual testing. *International journal of engineering and innovative technology*, 4(1), 2014.

[21] Advance Encryption Standard. Federal information processing standards publication 197. *FIPS PUB*, pages 46–3, 2001.

[22] Shane Tuohy, Martin Glavin, Ciarán Hughes, Edward Jones, Mohan Trivedi, and Liam Kilmartin. Intra-vehicle networks: A review. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):534–545, 2014.