POLITECNICO DI TORINO

Master degree course in Computer Engineering
(Embedded Systems)

Master Degree Thesis

Implementation of a Vehicle Function for an automotive Electronic Control Unit

External Temperature Management to avoid Overheating effects

Supervisor
Prof. Massimo Violante

Candidate
Piergiovanni Ferrara

Company supervisor
Mrs. Lorena Capuana

ACADEMIC YEAR 2017-2018

To my girlfriend for her support and love.
To my family too, who gave me the opportunity to study
to achieve this important goal.

Acknowledgements

I would like to thank Prof. Massimo Violante that gave me the opportunity to meet TXT e-solutions company and make a thesis work with their collaboration.

I would like to also thank my company supervisor Mrs. Lorena Capuana and other people of the company, for the provided support throughout the development of my thesis work.

Last but not least, a big thank goes to Mr. Devis Renna of a big electronic components maker for the automotive industry (Tier 1), who answered to all my questions related to the requirements of the project.

Contents

Summary

Significant researches over the course of the last years have contributed to the invention of new technological features installed in vehicles. They range from safety to entertainment features for which costumers are more than willing to pay. Many devices are involved in today's cars to warn the driver and show information about the environment like car rear view cameras, pedestrians detectors, lane crossing detectors, external temperature sensors, etc. Among all, modern cars have temperature sensors to measure and display the ambient temperature to the car dashboard for the driver. It is a very appreciated feature and despite appearing to be provided by a not so complex device, it has to be properly managed. This is because when the temperature sensor is installed on the side view mirror of the car, it is affected by the overheating effect caused by many possible events. In fact, the sun could directly irradiate the sensor or it could even receive the heat coming from warm surfaces of the vehicle itself or surrounding vehicles.

In this thesis work, a software solution to avoid annoying behaviours and unreliable temperature measurements has been developed. To achieve it, the logic implements sampling and filtering operations before sending the external temperature measurement as output.

Starting from the analysis of the requirements of a big Original Equipment Manufacturer (OEM), the External Temperature Management (ETM) *Vehicle function* has been developed by following the Model-Based Software Design (MBSD) ap-

proach which is highly adopted as a software development methodology in the automotive sector and allows to auto-generate a bug-free C-code. After integrating *TargetLink* tool, the C-code can be deployed into the target Electronic Control Unit (ECU). In addition, the software has been implemented to be AUTomotive Open System ARchitecture (AUTOSAR)-compliant in order to satisfy OEM requirements.

Many tests have been carried out to validate the software implementation, both on the model during Model-in-the-loop (MIL) phase and on the real BCM during the Hardware-in-the-loop (HIL) phase.

Introduction

In a technological age, a societal need becomes of interest for researchers that introduce new products, services and ideas to the market. For instance, the history of automotive technology and human factors research can be viewed similarly [15]. When we look at the last fifty years, we understand that being a mechanical engineer was enough, today most vehicle components are controlled by embedded systems and the new technological features we may currently have on a vehicle make growing the number of Electronic Control Unit (ECU)s , up to hundreds of them. These are the results of years of researches and studies in the automotive sector. In fact, many and many years ago the technology was not so advanced.

In the new age, it is widely-known that modern cars are always equipped with new devices and sensors to make our life easier, from safety-related electronics to entertainment features. Instrumentation panel (also called *dashboard*), for instance, shows traditional information but is also becoming more and more user-customizable.

The aforementioned ECU term refers to an embedded system able to control other electronic elements and receive information, within the vehicle, after downloading a software algorithm on it. Throughout the thesis work, the term *Body Computer Module (BCM)* will be also used to indicate an Electronic Control Unit in charge of controlling vehicle's accessories like power windows, power mirrors, air conditioning, immobilizer system, central locking, etc., as in this case. For the sake

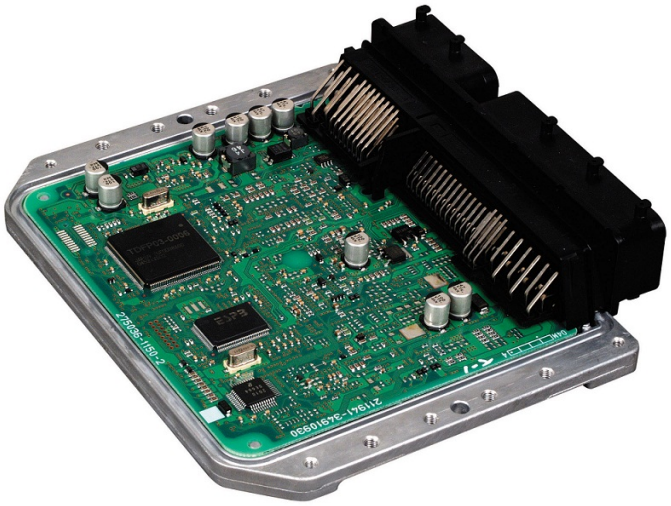

Figure 1: Typical vehicle ECU

of clarity, a typical ECU in shown in figure 1, while an ECUs network on a car is shown in figure 2.

Many ECUs are interconnected and each of them has been programmed to perform a specific function. They are placed in different locations and the communication between them relies on suitable communication buses to exchange important information within the vehicle. Clearly, one of the most appreciated info shown on the dashboard is the external temperature. In fact, nowadays every car owns temperature sensors that are able to measure the ambient temperature outside the vehicle and display it on the dashboard. Car makers may decide to install the sensor in different external locations on the car, for example, behind the bumper or the grill.

Some other car-makers are used to install the external temperature sensor on the side view mirror of car, because it could be a good place for hosting it. However, it is often affected by the *overheating phenomenon* that causes an overheating effect on the sensor. This occurs because some events can influence the real temperature value of the environment measured by the Negative Temperature Coefficient (NTC)

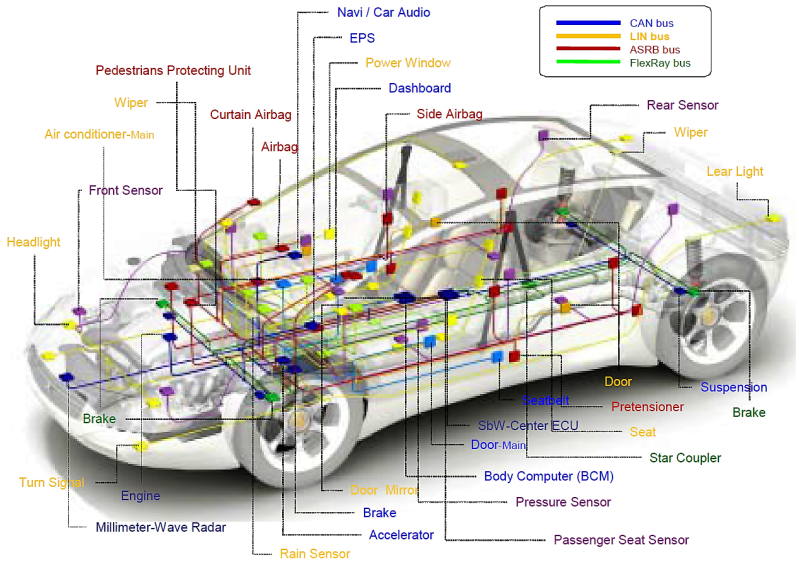

Figure 2: ECUs network and communication buses on a car [from [1]]

sensor. Possible events could be:

- Direct sun radiation towards the sensor
- Heat radiation coming from hot surfaces near the NTC sensor (car body, mirror cover, car engine, etc.)
- Heat radiation coming from hot surfaces even far from the NTC sensor (asphalt, etc.)
- Air convection due to warm air coming from near vehicles

In order to avoid unpleasant behaviours it is strongly necessary to filter the temperature measurements and evaluate the condition of the vehicle before giving the value as output.

This work aims at providing a solution to avoid incorrect and unreliable temperature values to display in the car dashboard for the user. The specifications of

what to achieve come entirely from a real project of a big Original Equipment Manufacturer (OEM). The goal is to implement a *Vehicle Function (VF)*, that is the software application implementing the algorithm for managing the external temperature. The VF will be developed by following a specific design approach, mainly used in the Automotive domain. The adopted approach is the *Model-Based Software Design (MBSD)*. Furthermore, as the software will be deployed on a real automotive *BCM*, developed by a big electronic components maker for the automotive industry (Tier 1), it must be compliant with the *AUTomotive Open System ARchitecture (AUTOSAR)* standard.

The main goal of the work is not only to provide a working solution, but also an optimized one, in terms of code occupation in memory and execution time for the Central Processing Unit (CPU), that mitigates the waste of hardware resources. To achieve it, the implementation will be conceived in such a way to make the software lightweight.

The presented thesis is divided into five chapters:

In **Chapter 1** the *AUTOSAR standard* is presented and its main concepts described, to give the knowledge for understanding how the Vehicle Function is structured to follow the automotive standard.

In **Chapter 2** the most common *In-Vehicle Network protocols* are described. Among them, Controller Area Network (CAN) protocol will be part of the developing work to exchange data.

In **Chapter 3** the *Model-Based Software Design (MBSD)* approach is explained. It is the used design method for the presented work and highly adopted in the automotive domain.

In **Chapter 4** the *External Temperature Management (ETM)* Vehicle Function is described following a top-down approach. Its behaviour and algorithm are explained. In addition, the flow concerning the model test, code generation and integration is shown.

In **Chapter 5** *Tests* executed on a real BCM and *results* of the proposed approach are analysed and reported.

Chapter 1

AUTOSAR

1.1 Introduction

AUTomotive Open System ARchitecture (AUTOSAR) is a worldwide development partnership of automotive interested parties founded in 2003. It pursues the objective of creating and establishing an open and standardized software architecture for automotive Electronic Control Unit (ECU)s (from [20]). The partnership structure includes: *Core partners (OEM & Tier1 supplier)*, *Premium Members* and *Associate Members*.

In the following sections, the standard will be briefly described.

1.2 Main working topics

In order to discern the concepts of the AUTOSAR standard, it is suggested to follow the description of the main working topics. They are:

- **Architecture**: three-layered Software (SW) architecture including AUTOSAR Application layer, Basic Software (BSW) and Runtime Environment (RTE)

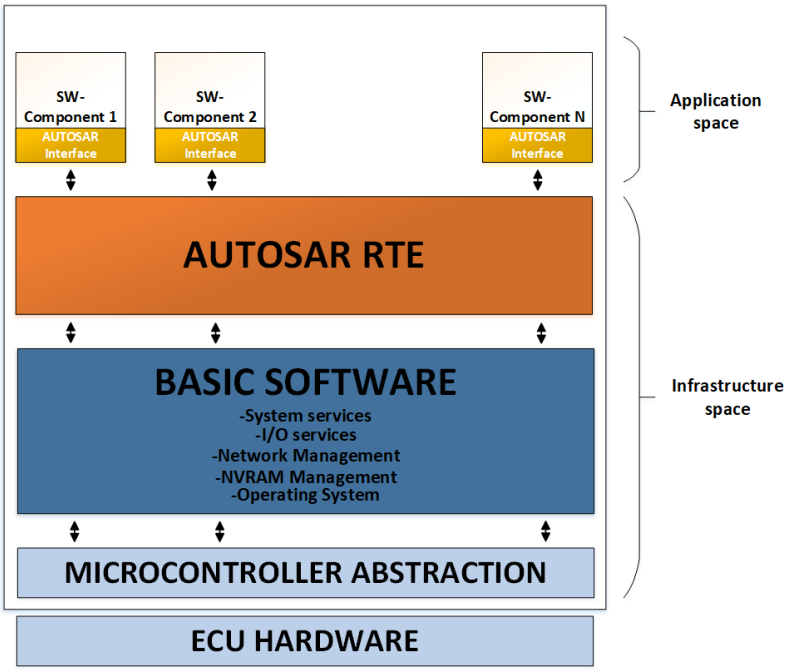

Figure 1.1: AUTOSAR architecture made by layers

- **Methodology**: formats and templates to include the configuration process of the BSW stack and integration of the application SW into the ECUs
- **Application interfaces**: interfaces specification for typical automotive applications in terms of syntax and semantics (as a standard for application software).

These topics are explained in the next sections.

1.2.1 Architecture

As written before, a typical AUTOSAR SW running on an ECU is made of layers, therefore the actual AUTOSAR architecture is shown in figure 1.1.

Basically, there are three main layers already mentioned before, however, the BSW layer can be further divided in many layers as it includes several services. It

is also fundamental the separation between the *Application* space and the *Infrastructure* space. The first contains the SW applications (topmost layer), while the second includes the remaining layers. A brief description on the main elements is provided in the following sections.

1.2.1.1 Software Component (SW-C)

The application space, in AUTOSAR, is made of interconnected SW-Cs, that represent the SW application, provided with an AUTOSAR interface and described by an Software Component (SW-C) Template.

An SW-C encapsulates part of the functionality of the application [6]. It means that we may have even more than one SW-C to make the overall SW application. In fact, an SW-C has well-defined ports, through which it can interact with other components [3]. A port is used to provide or retrieve information (data, operations calls, etc.). As already mentioned above, every component is defined by using an SW-C template, such that a new *Component Type* is also defined. The latter implies that such a component can be used an arbitrary number of times within the same system as well as in different systems, therefore multiple instances of the same component may exist [5]. Finally, components are developed against the Virtual Functional Bus (VFB), described later, without direct dependency on ECUs and communication busses.
To be an AUTOSAR SW-C complete and ready to be shipped, we need to have [6]:

- A complete and formal Software Component Description which specifies how the infrastructure must be configured for the component
- An implementation of the component, which could be provided either as *object code* (already compiled code) or *source code*

Concerning SW-C ports, AUTOSAR supports three main ports interfaces [3]:

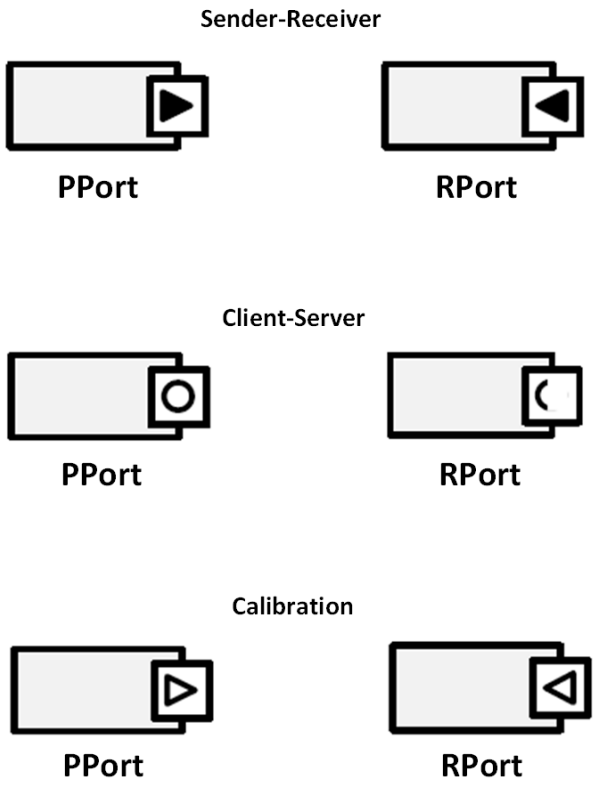

Figure 1.2: AUTOSAR-standard port-icons of ports interfaces

- Sender-Receiver: a sender distributes information to one or up to several receivers, or one receiver gets information from one or up to several senders
- Client-Server: a server provides a set of operations (Get, Set, etc.) and several clients can invoke those operations
- Calibration: static communication pattern that allows modules to access static calibration parameters.

They have its own port-icons according to the AUTOSAR standard:

In the figure above, for every port interface:

- *PPort*: refers to the port that provides something to others
- *RPort*: refers to the port that receives something from others

1.2.1.2 Virtual Functional Bus (VFB)

Virtual Functional Bus (VFB) is the most abstract level, where components are described with the means of datatypes and interfaces, ports and connections between them, as well as hierarchical components [12]. To achieve the *relocatability*, AUTOSAR SW-Cs are implemented independently from the underlying hardware [6]. This entails that components must not call directly the Operating System or the communication Hardware, therefore components can be deployed (integration process) to ECUs very late in the development process.

1.2.1.3 Run Time Environment (RTE)

Runtime Environment (RTE) is the implementation (for a particular ECU) of the AUTOSAR VFB and acts as a system level communication center for inter- and intra-ECU information exchange [6].

By specifying interfaces and their communication mechanisms (application dependent), the applications are decoupled from the underlying HW and BSW, enabling the realization of standard Library Functions.
In principle, the RTE can be logically divided into two sub-parts that realize:

- the communication between SW-Cs
- the scheduling of the SW-Cs

To fully describe the concept of the RTE, the *Basic Software Scheduler* has to be considered as well; the Basic Software (BSW) will be described in the next subsection. The Basic Software Scheduler schedules the schedulable entities of the basic software modules. Since an AUTOSAR Software Component is not allowed to access Basic Software directly, the access to services and other elements is abstracted via ports and AUTOSAR interfaces. With respect to the component implementation, therefore the RTE provides appropriately generated Application Programming Interface (API)s for Basic Software access.

1.2.1.4 Basic Software (BSW)

The BSW is the standardized software layer, which provides the infrastructure and services for execution of Software Components on an ECU as an integration platform for Hardware (HW)-independent software applications. Infrastructure is separated from the Application, in fact the latter is defined by the SW-Cs (see fig. 1.1).

BSW includes:

- System services
- I/O services
- Communication and Network management
- NVRAM management
- Operating System (OS)
- Microcontroller abstraction
- Complex Device Drivers, etc.

Essentially, the BSW aggregates all the functionalities that are utilized by the applications and actually point to resources at lower levels.

1.2.1.5 Microcontroller Abstraction layer (MCAL)

Access to Microcontroller registers is routed through the *Microcontroller Abstraction Layer (MCAL)* to avoid direct access to microcontroller registers from higher-level software. Thus, it contains internal drivers with direct access to the μC and peripherals, therefore handles the requests from BSW. In addition, MCAL implements notification mechanisms to support the distribution of commands, responses and information to different processes [6]. As you may imagine, its implementation strictly depends on the microcontroller.

1.2.1.6 ECU Hardware

This is the last layer that represents the actual hardware including the Microcontroller, so the CPU and its peripherals with their characteristics.

Hereinafter, to have a clear overview on how SW-Cs communicate and interact, some remarkable words are provided.
SW-Cs communication is supported by ports that implement the interface according to the communication paradigm (client-server, sender-receiver, ..etc). The communication is channelled via the RTE, and the proper layer in the BSW is encapsulated, therefore not visible at the application layer.
Hence, SW-Cs inside a single ECU are connected each other through SW-C ports and RTE, while different ECUs communicate through CAN bus (or others) via SW-C ports.

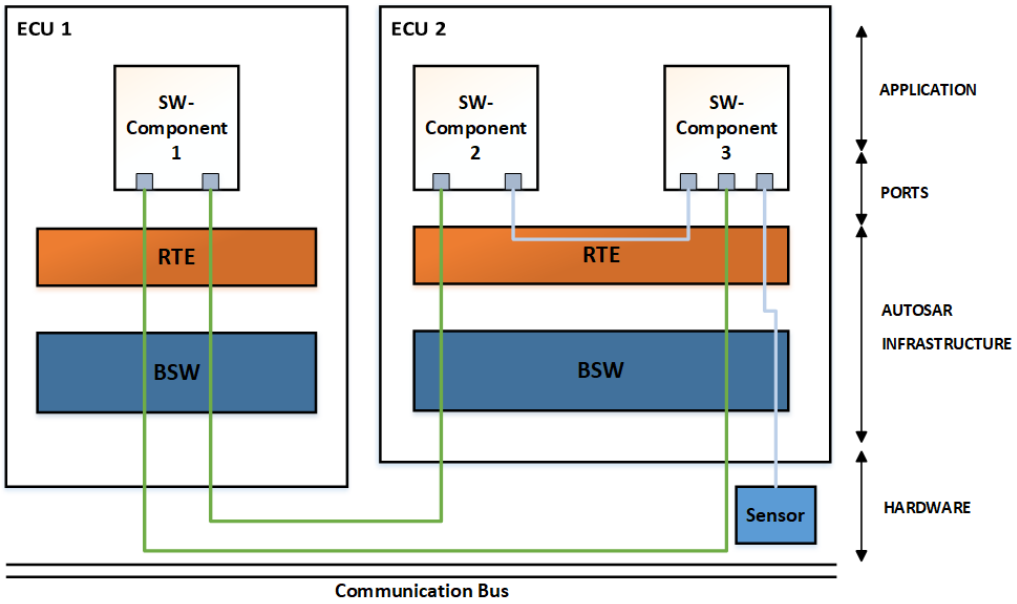

Figure 1.3: Communication between SW-Cs and ECUs

In the figure above, communications between SW-Cs within an ECU (grey lines) and between different ECUs (green lines) are clearly visible.

1.2.2 Methodology

Following the *AUTOSAR methodology*, the architecture is derived from the formal description of software and hardware components.
Starting from *templates* for *SW Component*, *ECU Resource* and *System* (with associated XML schema), a concrete specification is generated, presented as XML file and defined by OEM or Tier 1. Methodology prescribes to:

- Formally describe functional software in terms of SW-Cs description
- VFB takes SW-Cs description as input and validates the interaction of all components and interfaces before software implementation. VFB is implemented by the *AUTOSAR RTE* (it represents the concrete interface) and underlying

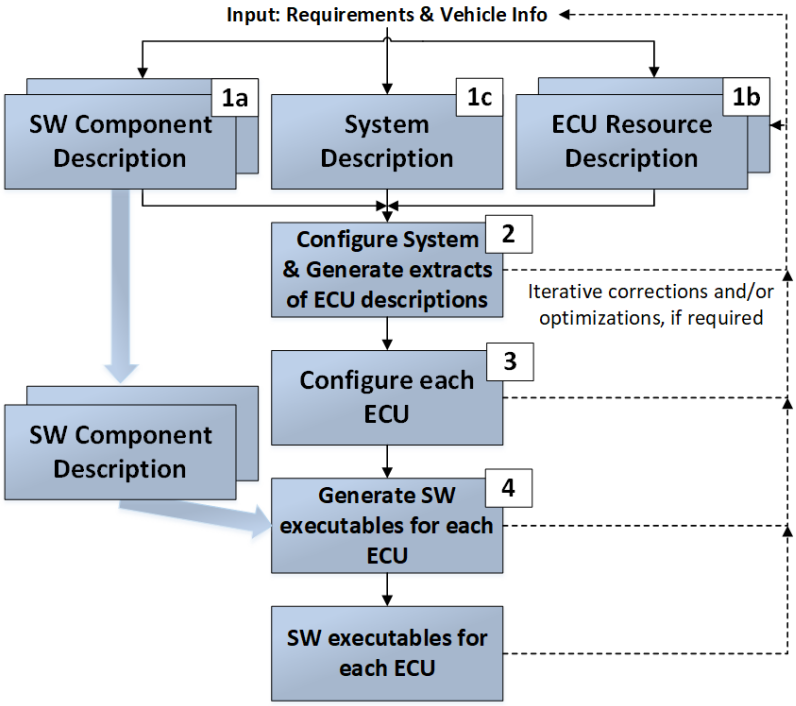

Figure 1.4: Software implementation process based on AUTOSAR-standard

BSW.

- Map of SW-Cs to ECUs and configuration of BSW

Hence, to configure the system, input descriptions of all software components, ECU resources and system constraints are necessary. The flow, that the AUTOSAR methodology prescribes, is reported in figure 1.4.

The implementation process follows those steps, starting from descriptions until the SW executables, for each ECU, are created. Steps are briefly described in the next sections.

1.2.2.1 SW-C description (step 1a)

The SW-C description consists on a formal description defined in a SW-C Template:

- General characteristics: name, manufacturer, etc.
- Communication properties: p ports, r ports and interfaces
- Runnable entities with trigger events, port access, etc
- Inner structure (composition): sub-components and connections
- Required HW resources: CPU time, scheduling and memory (size, type, etc.)

Structure and format are first described by the SW-C Template but then, what SW-Cs are connected together, as a system of SW-Cs, will be saved in one description which will be used for next process steps (mapping, software configurations, etc.).

1.2.2.2 ECU Resource description (step 1b)

The ECU Resource description contains:

- General characteristics: name, manufacturer, etc.
- Temperature: own, environment, cooling/heating
- Available signal processing methods
- Available programming capabilities
- Available HW: μC, architecture (e.g. multiprocessor)
  - memory
  - interfaces (CAN, LIN, FlexRay, etc.)
  - periphery (sensors / actuators)
  - connectors (i.e. number of pins)
- SW below RTE for micro controller

- Signal path from Pin to ECU-abstraction

The hardware is described independently of the application software.

1.2.2.3 System description (step 1c)

The System description contains:

- Network topology: bus systems, connected ECUs, power supply, etc.
- Communication
- Mapping/Clustering of SW-Cs

1.2.2.4 Distribution of SW-C descriptions to ECUs (step 2)

SW-Cs are distributed to ECUs. In particular:

- Configuration on the basis of descriptions (not on the basis of implementations!) of SW Components, ECU Resources and System description
- Consideration of ECU Resources available and constraints (for example, timing requirements) given in the System description

Figure 1.5 summarizes the distribution process to ECUs.

1.2.2.5 ECUs configuration (step 3)

Generation of required configuration for AUTOSAR-Infrastructure per ECU. It consists in taking the ECU description, System description and AUTOSAR RTE configuration information to build the AUTOSAR-Infrastructure. Figure 1.6 summarizes the ECU configuration process.

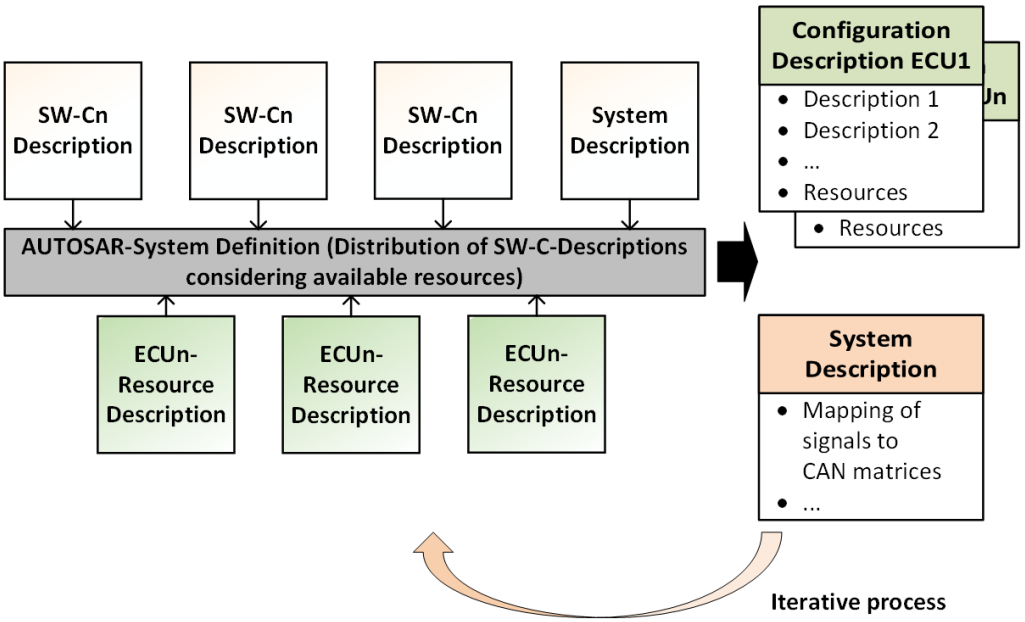

Figure 1.5: Distribution of SW-Cs descriptions to ECUs

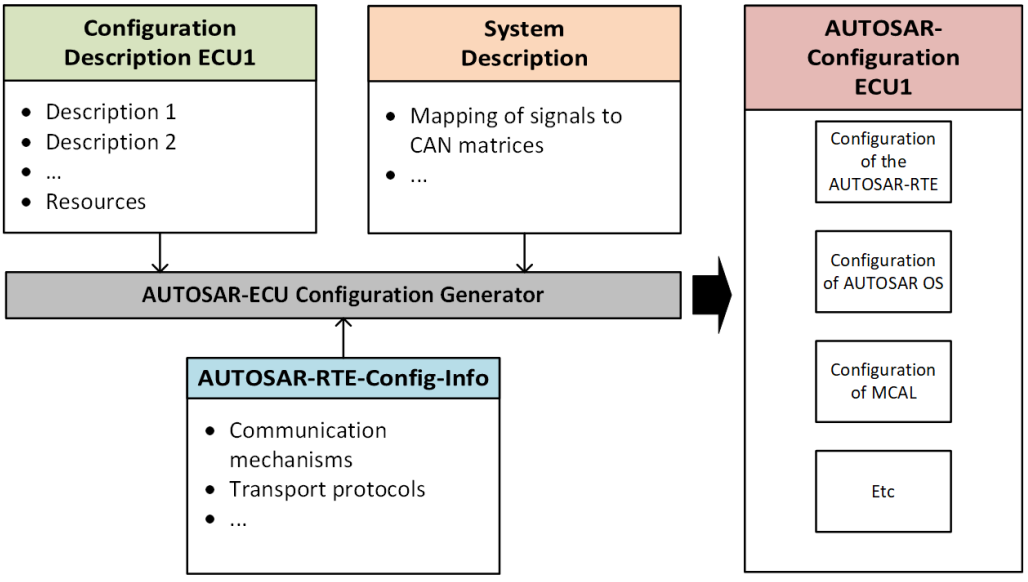

Figure 1.6: Configuration process per ECU

1.2.2.6 Software executables generation for ECUs (step 4)

After ECUs have been configured, the Software executables are generated for each and every ECU. This refers to the BSW,RTE and the linking of components ([6], [4]). The ECU Configuration Description is needed as it contains the information about which BSW modules and SW-C implementations are used to create the executable. The output of this step is the *ECU executable*.

1.2.3 Application interfaces

To ease the re-use of software components across several OEMs, AUTOSAR proceeds on the standardization of the application interfaces agreed among the partners. As described before, a SW-C has well-define ports interfaces to make the communication to other SW-Cs and among different ECUs. These standardized interfaces allow software designers and implementers to use them in case of expanding or reusing software components independent of a specific hardware and/or Electronic Control Unit (ECU). Typical examples of applications are electronic stability control (ESC), steering, electric parking brake, park distance control, exterior light, anti-theft systems, remote keyless entry and so on [2].

1.3 AUTOSAR benefits and drawbacks

Generally, the use of a standard as a reference for software development plays an important role, since it can provide lot of benefits. AUTOSAR benefits are:

- Use of common interfaces among OEM and Tier 1
- Increase in design flexibility
- Simplification of the integration task with the BSW

- Reduction of SW development costs
- SW-C exchange
- SW-C reuse for different HW platform

AUTOSAR standard embraces more than 180 organizations worldwide in the automotive domain who believe that this automotive standard really makes the difference, however some others are still no confident on it.

To capture a snapshot of the current benefits and drawbacks of using AUTOSAR, a web survey among the global AUTOSAR community has been performed [13]. Two research questions have been provided:

- Question 1: Which are the benefits of using AUTOSAR?
- Question 2: Which are the drawbacks and risks of using AUTOSAR?

Both questions had their answer options for what concerns the benefits and drawbacks about AUTOSAR.

Not surprisingly, the most mentioned benefit was *Standardization (88%)* that is the main characteristic of AUTOSAR software architecture, then *Reuse (80%)* and *Interoperability (51%)*; the others reached less than 50% of the common answers.

On the other hand, the most underlined drawback was *Complexity (65%)* and comments about it have been also provided by respondents. They have argued that AUTOSAR gets tougher when working with large projects and many developers, therefore they have suggested to make AUTOSAR more tool oriented in order to reduce that complexity. Then, *Initial investment (59%)* to teach it to people and *Learning curve (51%)*. The latter states that many engineers have difficulty in learning the standard.

Chapter 2

In-Vehicle Network Protocols

2.1 Introduction

Electronic Control Units (ECUs) are the most common components that control all the electrical and electronic parts of a vehicle. In the Introduction, we have seen that a modern car is surrounded by hundreds of ECUs where each and every of them performs a specific task (see figure 2). In addition, they exchange important data, get information from the environment by using sensors and drive several actuators.

But, how different ECUs, within the vehicle, are interconnected?

Many network protocols have been invented and developed for automotive purposes by big organizations. In the past, electronic devices were connected by using simple wiring systems, but then they were replaced to reduce the costs and weigh of them in the vehicle. The new most important network protocols are: *Controller Area Network (CAN)*, *Local Interconnection Network (LIN)* and *FlexRay*. They differ in some characteristics that suggest their usage according to the needs. Table 2.1 provides a comparison.

	Application	Bus Access	Control	Physical layer	Bandwidth
CAN	Soft Real-Time systems	CSMA/CD	Multiple master	Electrical	500 Kbps
LIN	Low-level Communication systems	Polling	Single master	Electrical	19.6 Kbps
FlexRay	Hard Real-time systems	TDMA/ FTDMA	Multiple master	Electrical/ optical	10 Mbps

Table 2.1: In-vehicle network protocols comparison

These network protocols must satisfy safety and security requirements when involved in vehicles. In the next sections, they will be briefly described.

2.2 Controller Area Network (CAN)

Controller Area Network (CAN) is a serial communication protocol originally developed by Bosch in 1985. It has been adopted for the first time in 1993 after becoming the international standard known as ISO 11898.

CAN applications

CAN protocol has been thought, at the beginning, to be used in the automotive sector, as it was suitable for making the communication between ECUs in vehicles. However, as other industries have realized the advantages that CAN offered over the past 20 years, they have adopted the bus for a wide variety of applications. For instance, Railway applications such as streetcars, trams, undergrounds, light railways, and long-distance trains incorporate CAN protocol. But that is not all,

CAN buses can be found in many aerospace applications, medical equipments in hospital, lifts and escalators, etc. [10].

For what concerns CAN in vehicles, it can be found on different levels, for example, in linking the door units or brake controllers, passenger counting units, and more.

CAN characteristics

CAN protocol was developed as a multi-master bus made of two wires only (*CANH* and *CANL*), in which data are not sent as large blocks from point A to point B under the supervision of a central bus master [11], but in broadcast to the entire network in order to guarantee data consistency in every node of the system.

It is a *Carrier-Sense Multiple-Access Collision-Detection (CSMA/CD)* communication protocol. CSMA means that every node must check the bus status before sending data on it and wait for a prescribed period in case of bus occupancy. CD means that collisions are detected on the bus and solved through a bit-wise arbitration, based on a preprogrammed priority of the message in the identifier field of a message.

How does the arbitration work?

During arbitration, every transmitter compares the level of the bit sent with the level read from the bus, when the levels are equal means that bus is free, therefore any node may start to transmit a message. A CAN frame/message, sent in broadcast, does not contain any address of the destination node, but an arbitration ID. It is unique and identifies every frame such that each CAN node may decide whether to get the message or not. When multiple masters start transmitting simultaneously, as soon as one master generates a dominant bit *(0)*, the other that transmits a recessive

bit *(1)* will lose the contention and will retry the transmission later. Thus, the node with the higher priority message gains bus access. This mechanism consists in a non-destructive bit-wise arbitration since the node can continue with the sending without destroying the message.

A typical CAN transceiver has a driver input and a receiver output, both pulled high internally. The driver input is used to send data on the bus, while the receiver output permits to read the bus status. Whenever the bus is free, i.e. no one is transmitting on it, a logical *1* (high voltage) is read; this is because of the lines that are pulled high. CAN transceiver schema is shown below:

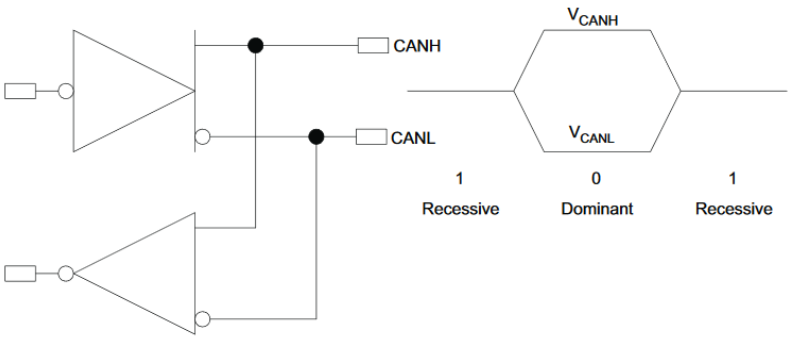

Figure 2.1: CAN transceiver components and inverted bus logic [from [11]]

On the top side of the figure 2.1 it is the driver input, while the receiver output is shown just below. The CAN transceiver is needed for interfacing the digital part to the physical CAN lines. In addition, as written before, the logic is inverted, therefore *0* is the dominant bit, whereas *1* is the recessive bit.

Finally, a CAN node, is reported in figure 2.2. It is composed of a microcontroller along with a CAN controller and a CAN transceiver.

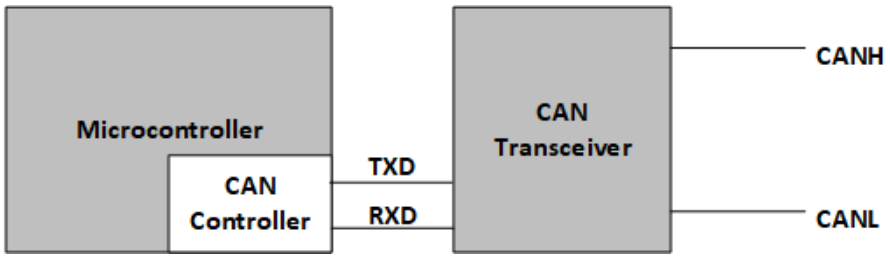

Figure 2.2: A CAN node

2.3 Local Interconnection Network (LIN)

Local Interconnection Network (LIN) is a serial communication protocol founded in the late 1990s by the LIN Consortium made of five carmakers: BMW, Volkswagen Group, Audi Group, Volvo Cars and Mercedes-Benz.

LIN applications

LIN procotol is suitable for automotive purposes, like CAN protocol, because it is inexpensive, easy to use and provides a good level of security. We can state that LIN is a cheaper *complement* of CAN as they are involved together. However, LIN is slower than CAN.

LIN characteristics

LIN protocol was developed as a single-master and multiple-slave bus made of one wire only. A good level of security is achieved thanks to different mechanisms like parity bits or checksum. The master is in charge of initiating a communication.

A LIN frame is made of a header and a response part (length from 1 to 8 bytes) [17]. The header part is sent by the master at the very beginning, then the response part, in case of data to be sent to the slave. If the master requests data from the slave, the latter sends the response part.

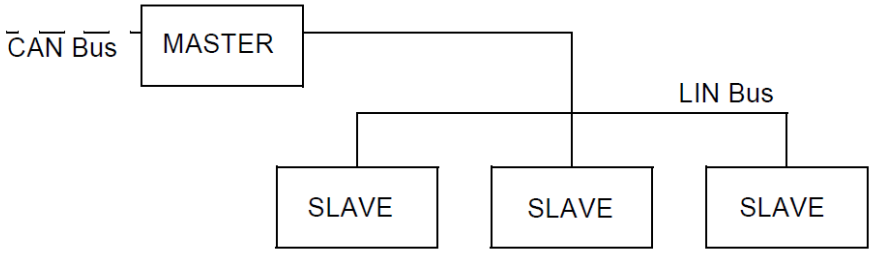

Figure 2.3: A typical LIN network bus [from [17]]

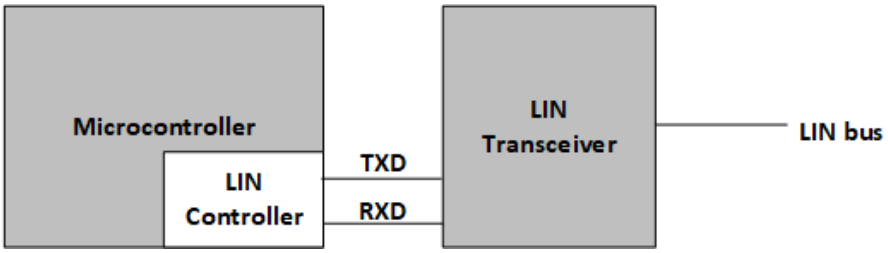

Figure 2.4: A LIN node

Like CAN protocol, LIN frames are sent in broadcast, therefore all the nodes get the same message. In addition, the frame does not contain an address. It is also important to note that LIN protocol does not allow a direct slave-to-slave communication and arbitration (unlike CAN) is not needed because there is a single master only. A typical LIN network bus is shown in figure 2.3.

From figure 2.3, it is worth noticing the co-existence of the CAN bus and the LIN bus. The first is mainly used to connect master devices since it offers a higher speed, while the second one makes the connection between a single master and many slaves (actually up to 16 allowed). Finally, a LIN node is shown in figure 2.4.

Even here, microcontroller requires a LIN controller, connected to the LIN transceiver to allow the translation from the digital part to the physical LIN bus.

2.4 FlexRay

FlexRay is a serial communication protocol developed by the FlexRay Consortium with the goal of obtaining a fault-tolerant and deterministic standard.

FlexRay applications

Main fields of application of *FlexRay*, like the other twos protocols, are the automotive sector and safety-critical applications [19]. This communication protocol is even faster and more reliable than CAN protocol. Its high data rate can achieve up to 10 Mbps.

FlexRay characteristics

FlexRay does not restrict the communication to any specific physical topology. It can be: point-to-point topology, passive start or active star topology.

A FlexRay frame is made of three parts: header, payload and trailer. The header consists of 40 bits and includes the ID, the payload consists of 254 bytes and the trailer of 24 bits.

Signals are physically transmitted based on the transmission of differential voltages, on Bus Plus (BP) and Bus Minus (BM)) lines) ([19]), to reduce possible interferences.

For what concerns the bus access, FlexRay nodes can obtain it in two different ways: Time Division Multiple Access (TDMA) and Flexible Division Multiple Access (FDMA) methods. TDMA consists in the division of the bus access in time slots of equal length, each assigned to every node, therefore all the messages are sent periodically and deterministically by the nodes. For sporadic transmission, the protocol actually allows to send event-driven and dynamic messages, this is the FDMA method.

A FlexRay node includes, like the others, a microcontroller along with a FlexRay controller and FlexRay transceivers (one or two bus drivers, depending on the number of channels). It is shown in figure 2.5.

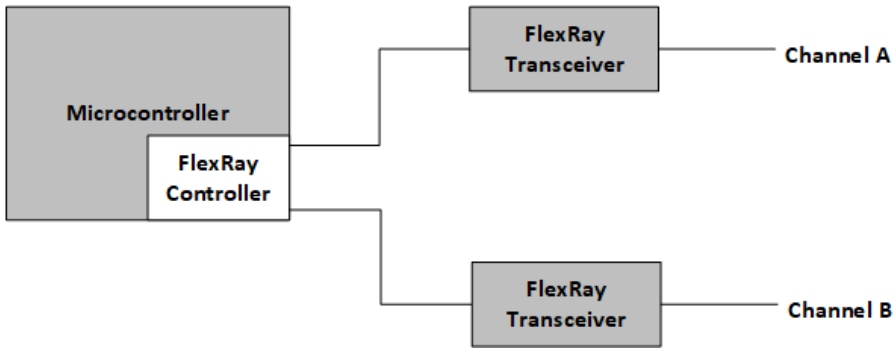

Figure 2.5: A FlexRay node

Chapter 3

Model-based software design

3.1 Introduction

The substantial growth in the number of vehicle Electronic Control Unit (ECU)s on average and in the complexity of the algorithms that reside on their controllers, has influenced the developing of embedded software, which is becoming more and more complex. As a result the number of code lines increases as well, also because of the amount of things the modern software has to perform. But then, the code has to be deployed on the target hardware that, as a consequence, could not host all of that.

The latest automotive environment also needs reusable software in order to be embedded in many different frameworks without so much difficulty. These aspects resulted in one of the most significant initiatives in the automotive industry in the years [16].

3.2 What is Model-based software design?

Model-Based Software Design (MBSD) methodology addresses that unmanageable complexity and is getting very used when developing embedded software. It is

mostly suggested when dealing with complicated and real-time control systems in many domains: Aerospace, Automotive, Industrial automation, etc. For instance, the modern Automotive environment is growing day by day with new hardware and new functionalities such that companies must be competitive on the market. MBSD is applied on safety systems, body controls, powertrain, etc. Even more, it can be found in the development of multimedia systems and entertainment. Although MBSD is highly suggested in complex designs to develop, the use of it is convenient also for less complicated designs because contributes in saving time and reducing costs.

Model-Based Software Design is used to define in a clearer way the system design specifications, to test system behaviour and to automatically generate the code for software production and rapid prototyping. In addition, it offers the possibility to validate in real-time the system before sending it to the manufacturing line.

What is a model? *A model is the graphical representation, by means of visual method, of software algorithms. For instance, in the automotive domain it consists in modelling Application Software Component (SW-C)s (see section 1.2.1.1) of functionalities for Body Computer Module (BCM), for which the AUTomotive Open System ARchitecture (AUTOSAR) open standard is followed.*

Traditional embedded software is made by hand coding and followed by verification sessions to check its correctness. These activities lack tool automation, so the human interaction is necessary. Unfortunately, this is error prone and time consuming too. The MBSD approach comes to the aid of the software developer for AUTOSAR-compliant applications and follows the V-shaped development flow. It helps in developing the software by adhering to the following sequential steps divided in two phases:

- Verification phase:
  1. System Requirements Analysis
  2. System Design
  3. Architecture Design
  4. Module Design
  5. Coding
- Validation phase:
  1. Unit Testing
  2. Integration Testing
  3. System Testing
  4. User Acceptance Testing

In a few words: starting from the analysis of the system requirements, the design of the system is produced based on the requirements (a further document may be created), without knowledge of the hardware. The system model also needs to be reviewed from a software implementation point of view before generating the code for the target hardware. While performing these steps, test plans and test cases are created at every stage, therefore errors and faults can be prevented and even detected earlier. In fact, the advantage of this work flow is that it allows to reduce delays and errors that could occur throughout the software development phase and for which clients could complain. [18]

In the next sections, each and every stage of the V-shaped approach is described, focusing also on the practical aspects of the MBSD, whereas at the end MBSD benefits are reported.

3.3 V-shaped development flow steps

The Software Development Life Cycle (SDLC) is a well-defined sequence of stages when developing a software product. There are different approaches that can be adopted, the V-shaped development flow is an extension of the Waterfall model. This is because in the latter, it is allowed to move to the forward stage only after completing the previous stage and it is not possible to step backward if something turned out to be wrong. On the other side, the V-shaped expresses the relationships between each development stage and the corresponding test stage. In fact, at every stage test plans and test cases are created for that specific stage. The V-shaped development flow is depicted in the figure 3.1.

System Requirements Analysis

First of all in the Verification phase, requirements of the system need to be collected. User's requirements concern interface, data, performance and security parameters that must be satisfied. During this phase, engineers want to understand what the desired system has to perform but not how to accomplish that (it is addressed later). In the meantime, the *user acceptance test* is planned and to gather information from the user, different methodologies are preferred like questionnaires, interviews, uses cases and others.

Functional Specification

After collecting system requirements, they have to be studied by engineers to get the purpose of the system in order to avoid that the client may complain. Documents are also produced to describe the intended specifications because at the end each requirement must be mapped to an element of the domain model.
At the same time, test plans and test cases for *system testing* are generated.

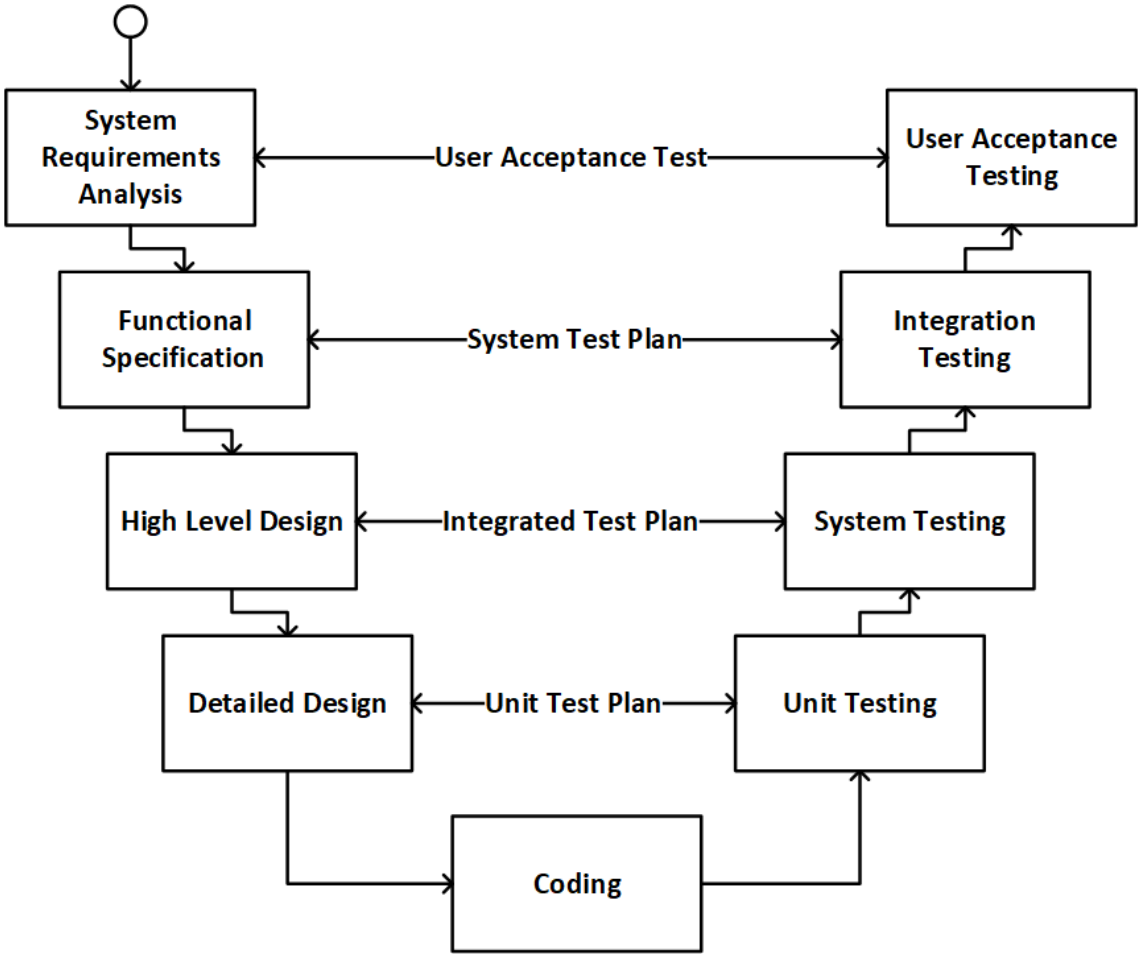

Figure 3.1: V-shaped development flow

High Level Design

Once the requirements have been correctly understood and functionalities specifications have been exhaustively highlighted, from a high level, the specific functionalities of the system are partitioned into submodules, which need to be designed and implemented later. Furthermore, as all the modules that represent system functionalities are interconnected together by means of interfaces, even their relationships needs to be tested. Thus, during this phase *integration testing* is also planned.

Detailed Design

Software functions have to be defined for every software module to implement the needed functionalities that engineers will develop. This is the lower level of design and as a support, a document may be very useful, in which all the elements including the interfaces, types and size are explained. Here, an *unit testing* is finally created.

Coding

After every functions have been defined, developers can actually start the coding phase. It consists in developing C-code for the system functionalities that must satisfy the user, according to the requirements documents. MBSD approach offers the possibility to autogenerate the C-code for the system model.

Unit Testing

The purpose is to test the functionality of every unit and to check its compliance with the specifications. A software unit is made of source code that needs to be all covered during the test. Basically, every code line have to be tested by means of test cases and test coverage has to be, as much as possible, the highest. Metrics for unit testing are: *Statement Coverage (SC)*, *Branch Coverage (BC)* and *Modified*

Condition/Decision Coverage (MC/DC). These are based on the requirements for the inputs of a software unit. There are also other suitable methods for unit testing, like *Interface test*, *Function injection test*, *Resource usage test* and *Back-to-back test*.

Integration Testing

After every unit has been tested, we want to confirm the interoperability of the units. Thus, test cases have to test the interaction of them. Used methods are very similar to those adopted for unit testing. Metrics for integration testing are: *Function Coverage (FC)* and *Function Call Coverage (FCC)*.

System Testing

Here, you need to define a set of test cases that can test the system when interacting with the environment. Hence, the software has to satisfy the requirements. Three different methodologies can be pursued: *Hardware-in-the-loop (HIL) testing*, *ECU network environment* and *Vehicle-in-the-loop (VIL) testing*. The first two are heavily used to figure out most of the problems in the software, whereas Vehicle-in-the-loop testing is strongly suggested when testing automated driving features because the software runs in a real vehicle with an emulated environment.

User Acceptance Testing

The purpose is to check that the system satisfies system requirements that have been analysed at the first step. The test is carried out by the user and does not involve the knowledge of the implementation, therefore the system is tested as a black-box. In general, the system functionality is tested.

3.4 MBSD - practical perspective

From a practical viewpoint, the engineering process is described in the next sections, in order to understand how engineers in their companies develop embedded software by following the Model-based approach.

3.4.1 Modelling

The process starts with the modelling phase when the system engineers create models of the system and environment [8]. In fact, they capture and follow the requirements that affect the environment and system's behaviour.

The usual adopted modelling software to create models is *MatLab*, provided with *Simulink* tool. The latter allows to build models for the *Plant* and the *Controller*, represented graphically, after analysing the requirements. On one hand, plant model represents the environment, therefore the surrounding elements of the controller model. The former can be created by exploiting Simulink blocks and mathematical equations to represent the plant dynamics. On the other hand, controller model, after analysing the plant and control requirements, can be created by making a finite state machine for which StateFlow is strongly suggested.

3.4.1.1 What is Stateflow?

Stateflow is a software tool, integrated into MatLab, that permits to create a finite state machine. The latter is a representation of an event-driven (reactive) system. Since it is reactive, the system makes a transition from one state to another, if the condition defining the change is true. A finite state machine can be used to represent many systems in the automotive sector for control-systems in cars. Starting point is the Stateflow Chart, with its inputs, outputs, that can contain sequential and combination logics in the form of state transition diagrams, flow charts, state tran-

sition table and truth tables. The Chart can be included in a Simulink model to build the so-called Stateflow machine. Local and constant variables can be defined in the Model explorer and they will be visible in every state of the Chart and can be used to define transition conditions. A Stateflow Chart may contain several states. A state describes an operating mode of a reactive system and it can be active or inactive, according to conditions and events. In addition, state hierarchy is also possible - it allows to organize hierarchical states with multiple levels in the Chart. Every state which contains multiple states is called Superstate. Another aspect to consider is the State Decomposition that differs in: Exclusive (OR) and Parallel (AND). On one hand, the state decomposition Exclusive (OR) of a superstate implies that all its substates are Exclusive (OR) as well and in particular it means that inner states describe operating modes which are mutually exclusive, therefore only one substate can be active at a time. They graphically have solid borders. On the other hand, the state decomposition Parallel (AND) describes concurrent operating modes and all the substates are active at the same time. In this case, they are graphically represented with dashed borders. Furthermore, it is also possible to set the main Chart as Parallel (AND) which indicates that the main superstates are concurrently executed, while being individually set as Exclusive (OR) (this implies the same behaviour to their substates).[14]

During the Modelling phase, engineers can also import the *arxml-file* [16], already prepared with the support of other tools, to generate a skeleton Simulink model that contains the interface blocks (inports and outports) and other elements as defined in the AUTomotive Open System ARchitecture (AUTOSAR) SW-C description file (see section 1.2.1.1). Plant dynamics have to be matched, therefore the controller needs to be designed accordingly.

Plant model could be actually already provided or simulated by using hardware simulator (last development stage). Thus, the main purpose is to model the Con-

troller. Once the model has been designed, we can connect Plant and Controller input/output ports.

3.4.2 Model validation

It is widely-known that it is easier to solve problems (correct bugs in the model) early in the development phase, therefore it is highly suggested to test and validate the created model before moving forward to the next step. For instance, testing the software of safety-critical systems, where damages can cause injuries, is crucial.

Testing usually includes the process of executing a program or model (in case of MBSD approach), with the intent of finding software defects; they could result in an error that may propagate to the end user if it becomes a failure and we want to avoid this bad inconvenience [9]. In alignment with research studies, we can state that MBSD approach for software safety is broadly applied in various application domains.

For that reason, once the models have been created, they have to be analysed. This phase is fundamental in the development process because it must provide assurances that if the system is built as described by the model, and the assumptions about the components, the infrastructure, and the environment are true, then the system will work as expected, i.e. exhibits the desired behaviour [8].

The basic idea of testing for MBSD is that instead of creating test cases manually, a selected algorithm generates them automatically from a model. One of the advantages of MBSD is that it allows tests to be linked directly to the system requirements which renders readability, understandability, and maintainability of tests easier [22]. Furthermore, it has been shown to provide good coverage of all the behaviours of the system and to reduce the effort and cost for testing.

Controller behaviour has to be validated with the respect to the specifications and refinements can be applied whether something turned out to be non-compliant

or wrong. This phase is also referred to as *Model-in-the-loop (MIL)* validation. It is based on the behavioural model of the system itself and is tested in an open loop (without a plant model, so simulated inputs) or closed loop (with a plant model).

Analysis and simulations, based on the models of the system, are carried out by using Simulink tool that allows to track signals and information exchanged between Controller and Plant; also we check how their states evolve. An important point in model-based embedded systems is that the analysis perfectly predicts the system's behavior, without actually building the system. However, this may not be feasible because of inherent inaccuracies in the models or lack of knowledge about details. But, we shall aim for, as a minimum, analysis techniques that will pinpoint potential problems in the design when implemented on the platform and give an early indication, well before integration [8].

Eventually, model is checked for compliance with modelling rules, such as *MatLab Automotive Advisory Board (MAAB)*, and naming conventions.

It is worth noticing that MBSD approach helps engineers in developing high quality and bug-free software with the support of specific and automated tools.

3.4.3 Code generation and integration

When the system design has been completely verified and validated, next step consists in generating the ingredients of the model implementation to be integrated in the target hardware.

The first step is known as *Synthesis* and is accomplished by generating the code for the target hardware (*Electronic Control Unit (ECU)* in vehicles). This can be done automatically because supported by many tools like Embedded coder (integrated in MatLab/Simulink), TargetLink (by dSpace), etc. The second step is related to the deploy of the auto-generated code into the target hardware.

However, before generating the code, Simulink solver type and time solver must

be set. Usually, fixed-step type of discrete time solver is chosen since we need to have a predictable number of model evaluations with the same rate to guarantee the real-time behaviour. Besides that, hardware implementation and code generation options need to be set-up as well. At this point, once the C-code has been auto-generated, tools allow to perform a Software-in-the-loop (SIL) validation that is the execution of the code together with the plant model, if any, in the development Personal Computer (PC). During this phase, we can also do optimizations while solving possible errors.

When we are sure that the implemented code does not contain any bug, the software application needs to be integrated into the BSW, which provides the services and drivers to communicate with the specific hardware.

Then, we are ready to deploy the code to the target hardware. We can validate the behaviour of the generated code by running it on the hardware, while the plant model still runs on the development PC with the simulation tool. It is called Processor-in-the-loop (PIL).

Last possible step can be the validation of the Real Time (RT) behaviour of the Controller even after being downloaded into the ECU that runs the control algorithm and hardware simulators emulate the environment, physically connected to the ECU, performing Hardware-in-the-loop (HIL), ECU-network environment or Vehicle-in-the-loop (VIL) validation activities.

3.5 MBSD benefits

Why should we use Model-based software design?

The choice of adopting this approach provides benefits ([18]) like:

- Improved product development process

- Better cooperation between software developers as a team
- Link of the system requirements between model and requirements document
- Early testing to get more confidence with the design and fix errors
- Software safety testing
- Production of high quality software (bug-free code) at acceptable cost
- Auto-generation of code and documentation
- Reuse of the software for other hardware platforms

However, model-based development is not without risk. It is not obviously clear whether a seamless development process from early design to final target code is feasible: Some design steps might demand knowledge of environment properties which are difficult to formalize. Design steps in the later phases will require precise knowledge of the target platform, for instance to access device drivers or in order to estimate the worst case execution times which are needed as input for scheduling algorithms [7].

Chapter 4

External Temperature Management (ETM)

4.1 Introduction

As already mentioned in the work introduction, I want to provide a solution that avoids the overheating effects and gives correct and reliable temperature measurements.

The main goal of this work is to implement an optimized software as a project for a big Original Equipment Manufacturer (OEM) in the automotive sector. *Code occupation* and *execution time* will be taken into account when designing the vehicle function. This is particularly crucial, since automotive Electronic Control Unit (ECU)s are equipped with microcontrollers that do not have infinite availability of resources (CPU and Memory), therefore in this sense it is necessary to optimize. In general, there is actually a big incoming request from the automotive industries about that.

4.2 Developed Vehicle Function (VF)

The *Vehicle Function (VF)* is called *External Temperature Management (ETM)* and the proposed implementation for the *ETM* follows the *Model-Based Software Design (MBSD)* approach (*chapter 3*) and *AUTomotive Open System ARchitecture (AUTOSAR)* standard (*chapter 1*) to develop a software to be deployed on real BCM.

Sometimes during the code writing phase, useless and not optimized elements can be introduced by developers and even programming errors can be made. On the other side, by exploiting the MBSD approach, you can avoid making code errors because the model, that represents your final software, can be validated many times and applying refinements is always possible. In addition, it is possible to conceive the model as much optimized as possible. Thus, during the development phase you get more confident with the model and eventually a more optimized and bug-free software is going to be generated.

Code generation can be automated after finalizing all the required checks on the model and guarantees the absence of bugs in the code.

Another aspect to be considered is that when developing automotive application the standard must be always followed, therefore the implemented software will be AUTOSAR-compliant. This allows a faster integration with the lower software layers and a higher versatility of the developed automotive application. The latter can be deployed without any difficulty into different ECUs that could even have different hardware inside. For more information, see the aforementioned AUTOSAR chapter.

Which is the usual product development process in the automotive industry?

In the most common cases, design information about the software we want to develop are written with the means of documentations. A documentation reports all

high-level and low-level requirements that the application must satisfy. In general, the most automotive industries follow the same flow. The Original Equipment Manufacturer (OEM) is the automotive manufacturer that represents the brand of the vehicle and is in charge of specifying the requirements, in a document, for the software he wants to obtain. Thus, in this process the OEM represents the customer. After the specifications have been described and the high-level design has been produced by the design team, the suppliers can start developing the software for the OEM (actually sometimes the OEM can develop part of the whole software). OEM suppliers are called Tier 1s, which supply hardware components but also software for vehicles. Further companies can come into play which behave as consultants to help and provide services to Tier 1. Documentation, sometimes not so understandable, is read and interpreted by a special team in the Tier 1/Consulting company, in order to clarify the algorithm and mistakes, if any. Then, the software is ready to be developed by software developers who try to follow step by step all the project requirements. At the end, before sending the software to the OEM, it must be validated by the validation team to guarantee customer expectations and correct behaviour.

The above concept has been followed for making the present work. Me, as a software developer for the consulting company *TXT e-solutions*, I have analysed the requirements document produced by a big OEM and provided the software implementation of the desired VF with the support of a Tier 1. The latter, as manufacturer of ECUs and other vehicle parts, provided me the possibility to execute tests on a real ECU.

In the next sections, firstly the *Functional diagram* is provided. Secondly, the general *Model architecture* is given with a top-down approach, such that we can move deeper into the main states to describe their functionalities. While modelling the system, test cases have been generated to get confident with the system model,

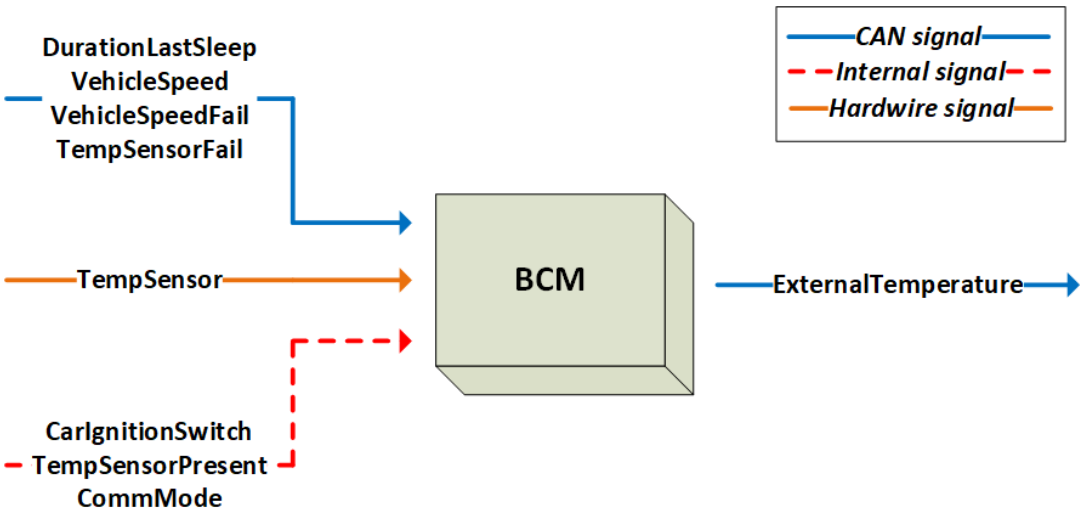

Figure 4.1: BCM functional diagram for the ETM vehicle function

and eventually code generation for integration has been performed.

To not disclose sensitive information owned by the OEM company, real variable names and some low-level details will not be provided.

4.2.1 Functional diagram

From a high level, the functional diagram for the BCM interfaces is shown in figure 4.1.

The BCM, which is nothing but an ECU used for this particular task, has its own input and output signals (see the direction of the arrows). They are routed by using different interfaces:

- *Controller Area Network (CAN)*: 2-wire communication protocol with high data rate (refer to sec. 2.2)
- *Internal*: internal (SW or physical) connection for data exchange

- *Hardwire*: wiring connection between components/units

The use of input/output ports will be explained in the following sections.

4.2.2 Model architecture

As already mentioned before, this work comes from a real project of a big international company that designs components for the automotive sector and we want to achieve a working and optimized software.

The actual model architecture for the ETM vehicle function follows exactly what the specification of the VF prescribes. The specification of the VF is written on an official document and an excel file is also provided. The latter helps to understand the external interfaces (ports) of the BCM and other information - it is updated whenever there are modifications on the requirements of the vehicle function. The final software, that requires to be validated in real-time, will be deployed into a BCM that receives the External Temperature value, as a discrete signal, from the *NTC temperature sensor* and other useful information from BCMs connected through the CAN communication protocol (see fig. 4.1), that is one of the most used protocol in vehicles.

What is a NTC sensor?

A NTC (Negative Temperature Coefficient) sensor is basically a resistor whose resistance varies significantly with the temperature and so takes the name of thermistor. It is made of a semiconductor material and can be used as a temperature sensor since the resistance decreases when temperature increases. The resistance can be measured by providing a known direct current through the thermistor to measure the voltage drop. The lower the voltage drop, the higher the temperature.

Input and output ports have to be consistent with the environment the BCM will be integrated on. Furthermore, ports and signals naming is also guaranteed in order to ease the cooperation between the application layer and the basic software layer. In fact, you may wonder that we can create an application without worrying about the hardware to use and their compatibility.

In terms of AUTOSAR-standard terms, developing a model for an automotive application means implementing a *Software Component* (SW-C, section 1.2.1.1) that will be executed cyclically or in response to some events by the Operating System (OS) as a *Runnable entity*. In this case, the Runnable entity will be executed periodically with fixed period $T = 50ms$.

To build a model for the External Temperature Management (ETM) vehicle function, *MatLab/Simulink* and *StateFlow* have been used. The second tool allows to create a finite state machine, inside Simulink, that evolves according to incoming inputs and local variables. Starting from a Simulink *.slx* or *.mdl* file, a StateFlow Chart can be created that contains states, conditions and transitions to make actions. Then, it is necessary to import the *.arxml file* that contains the interfaces blocks and other elements as defined in the AUTOSAR SW-C description file (see section 1.2.1.1). This allows to use the naming convention adopted in the specification of the VF.

It is worth highlighting that before actually implementing the model from scratch through the Matlab environment, it has been conceived and designed by hand on papers in order to understand how to deal and face the intended logic and algorithm. This is particularly important since the goal is to implement an optimized software. In fact, when building the finite state machine, it is suggested to avoid the use of redundant states whenever they can be merged together and useless Matlab functions. The latter would be translated into C-functions that for sure would decrease performance due to context switches. In addition, variables use has been controlled to the essential to reduce the memory size. The implemented StateFlow main chart

is reported in figure 4.2.

Input and output ports are clearly visible and names also suggest which is the functionality they provide. Names are the same of those shown in figure 4.1 and port types follow the current implementation. Despite that, their meaning is explained in the next section (4.2.3).

4.2.3 External interfaces

The main chart, therefore the SW-C in general, has input ports that can be differentiated in *DataReceivePorts* and *ClientPorts*, while the output ports are called *DataSendPorts*. To understand how they are used, according to the AUTOSAR standard, refer to section 1.2.1.1. They are highlighted in the tables below, provided with a short description to understand its specific usage.

DataReceivePorts are:

PortName	Description
TempSensor	Used by the BCM to get temperature measurement from the sensor
TempSensorFail	Used to know whether there is a fail on the temperature sensor or not
CarIgnitionSwitch	Used by the BCM to get the key mode status
VehicleSpeed	Used by the BCM to get the average vehicle speed
VehicleSpeedFail	Used to know whether there is a fail on the average vehicle speed or not
DurationLastSleep	Used to know the duration of last sleep state and to be added to the current time permanence in KEY_OFF

Table 4.1: Table containing SW-C DataReceivePorts

While *ClientPorts* are:

PortName	Description
TempSensorPresent	Used by the BCM to get whether the temperature sensor is present or not
CommMode	Used to know the communication mode (CAN Network)

Table 4.2: Table containing SW-C ClientPorts

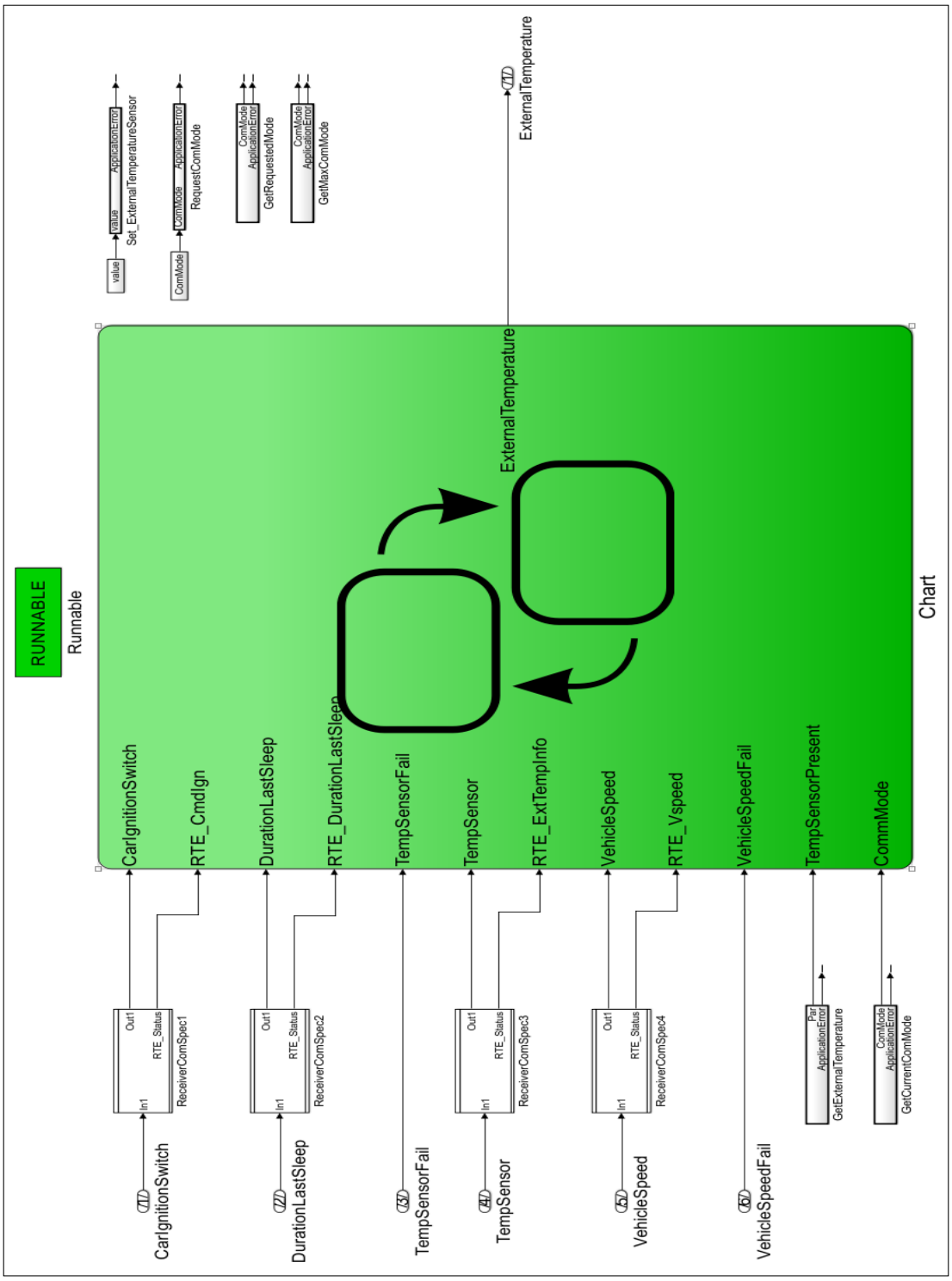

Figure 4.2: Main chart containing working states

Finally, there is only one *DataSendPort*, that is:

PortName	Description
ExternalTemperature	Used by the BCM to send the temperature value as output

Table 4.3: Table containing SW-C DataSendPorts

These are the main ports that the model uses to receive data from its inputs. However, as shown in figure 4.2, there are four more ports called *RTE_xxxx* which are used to know the Runtime Environment (RTE) returned value when getting the associated data. They will be used during the simulation phase.

4.2.4 Inner states

The model is periodically evaluated at a fixed rate of *50ms* by the Simulink solver, as requested in the specification of the Vehicle Function (VF).

The two basic inner states are *SamplingFiltering* and *TemperatureManagement* that are executed at the same rate. These states need to run concurrently and from the StateFlow point of view is necessary to set them as *Parallel(AND)*. However, the Simulink solver evaluates the states in sequence, in fact an execution order has to be decided in any case, therefore the concurrency is only an illusion for the software developer. But it is necessary in order to indicate that those states are independent one from the other and there is no hierarchy in the execution. Independent states are represented by Simulink with dashed lines and are shown in figure 4.3.

The execution order is clearly visible, in fact SamplingFiltering state shows the number 1, while TemperatureManagement state shows number 2. In the next sections, they are described.

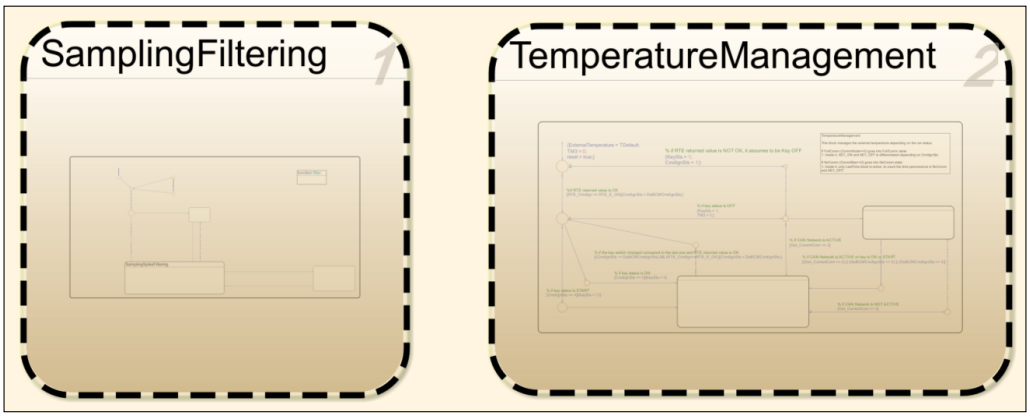

Figure 4.3: Inner states of the main chart

4.2.4.1 SamplingFiltering state

Focusing on the ETM specifications and the needed logic, on one hand the *SamplingFiltering* state is in charge of sampling the temperature values coming from the NTC sensor every *50ms*, putting them into a vector of 10 elements with a shift operation. A filtering operation is also fundamental to reject maximum and minimum values that sometimes may be spurious. Then, the arithmetic average, on 8 samples, is performed to obtain a reliable temperature value even though sharp temperature variations are very unlikely.

Values for the input port *TempSensor* come as a voltage measurement, but they will be translated into Celsius degrees by other software components. The allowed range for temperature is [*-54.5*, *99.5*] ℃. Inner states of SamplingFiltering state are provided in figure 4.4.

There are three states only inside. The main one is *SamplingSpikeFiltering* which performs the sampling and filtering of the input temperature values. To accomplish these operations, two Stateflow functions (*sample* and *filter*) have been implemented. The other states are: *KEY_OFF1minCount* and *No_Sampling*; the first state counts up to one minute when at KEY_OFF in order to sample every minute, whereas the

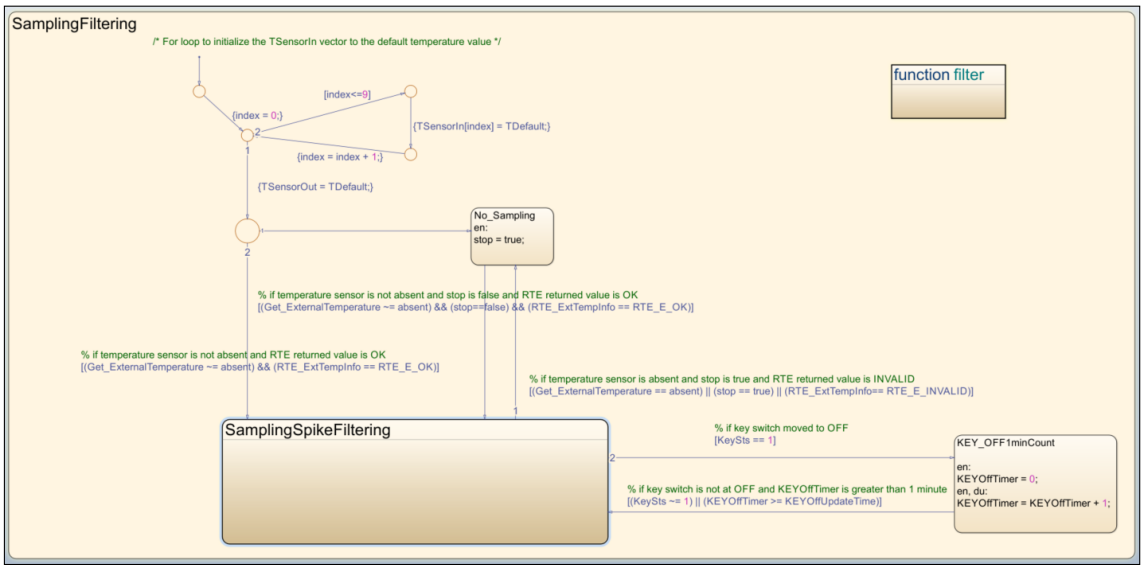

Figure 4.4: SamplingFiltering state blocks

second state is reached when something wrong happened, like temperature sensor not present or failed.

4.2.4.2 TemperatureManagement state

On the other hand, the *TemperatureManagement* state is more complex because it is the core of the logic that manages the temperature, before sending it as output. For the sake of clarity, its rationale will be explained in the next sections.
Before moving forward, I want to underline that the BCM will receive and send data through the CAN Network. The latter can be actually active or not and it generally depends on the *car ignition switch*. There are functionalities that work in *Full Communication (CAN Network active)* only, others work also in *No Communication (CAN Network not-active)*.

The core of the temperature conditioning process is active in Full Communication only.

At this point, it is worth introducing the concept of *car ignition switch* to un-

derstand what will be provided in the next sections.

What is a car ignition switch? *Car Ignition switch is the device in which we can insert the car ignition key. Generally, it has four positions to decide which car's electrical and electronic systems will be switched ON and OFF.*

First position is Accessories for which you can usually use the radio and regulate power window without starting the car's engine. Second position is OFF. It is referred to when the engine is OFF while some electrical systems still work. However, most of the elements are switched OFF in this ignition position. Third position is ON. This position starts all the electrical systems that can be used by the driver. This is the utmost position before starting the car. The last position is just the one that starts the engine. Once the driver turns clockwise the key and the car actually starts, the driver can release the key and it will return to its previous ON position.

For what concerns the ETM vehicle function, the only necessary and considered ignition status are: *OFF*, *START* and *ON* (also called *RUN*). The latter two can be merged together as they implicate the same condition for our model and we can also build an optimal model. In alignment with what has been mentioned before, we can say that basically the CAN Network is active at KEY ON and KEY START, while it becomes inactive after some point at KEY OFF whether there are no coming input data.

In the next sections, I will describe the functionality by differentiating the two macro-states *Key ON* and *Key OFF*, belonging to TemperatureManagement, in which the system will behave differently.

Key ON state

Key ON state, as the name suggests, contains functionalities that work only

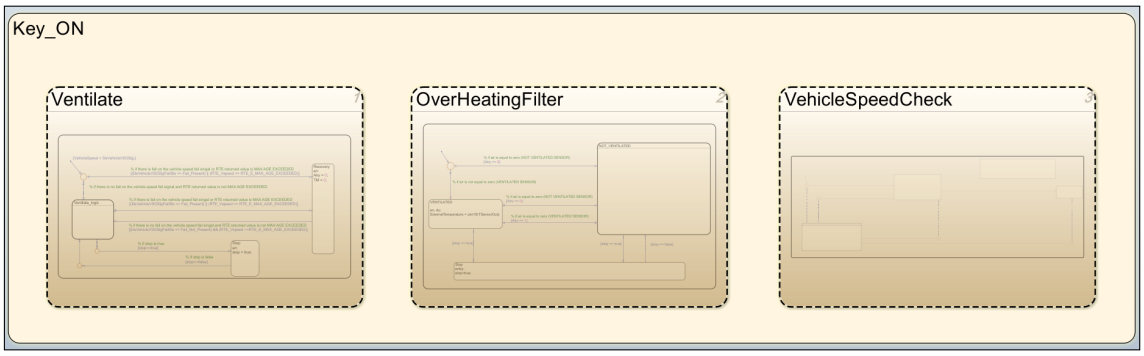

Figure 4.5: Key ON blocks

when the car ignition switch is moved to the third position and/or is maintained in that position. Main block are reported in figure 4.5.

Inner blocks *Ventilate*, *OverHeatingFilter* and *VehicleSpeedCheck* are independent each other.

The purpose of Ventilate state is to understand whether the NTC temperature sensor is ventilated, that is when the vehicle speed (DataReceivePort through CAN Network) is above the threshold (30 Km/h), or not. Whenever the sensor is considered as ventilated, then the ExternalTemperature output is updated, no matter the measured temperature value by the sensor. To make such a decision, the vehicle speed must be greater than the threshold for at least 60 seconds, called *T1* for the sake of simplicity. If the vehicle speed suddenly drops below the threshold while the timer did not reach *T1* time yet, the not-ventilated condition is considered again. Instead, if it drops while being in the ventilated state, the system must move to the not-ventilated state if the condition (vehicle speed $<$ threshold) is true for another predefined time of 30 seconds, called *T2*. This mechanism is managed by the *Ventilate* block. It is a fundamental algorithm and its high-level logic is provided in figure 4.6.

OverHeatingFilter block checks in parallel whether the ventilate condition (set by the Ventilate state) has been achieved or not, and updates the ExternalTemperature

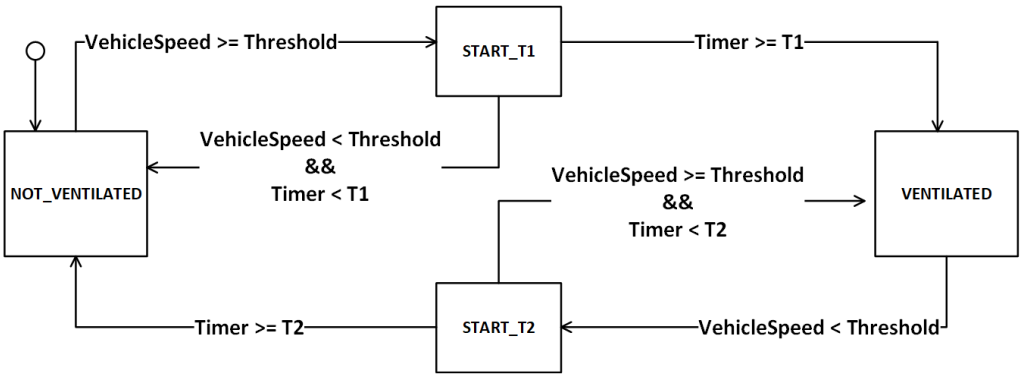

Figure 4.6: Ventilate logic: updates the temperature sensor status as ventilated/not ventilated

accordingly. Thus, this block is the only one in charge of changing the output value while at Key ON

When the sensor is known to be ventilated, the ExternalTemperature is updated. Instead, if the sensor is not ventilated and the measured temperature value is greater than the current ExternalTemperature value, the latter is not updated since a hot condition for the sensor is supposed, otherwise it is updated (measured temperature $<$ current ExternalTemperature). Of course, the output must belong to the same allowed range of the input, but it is managed by another block.

Last but not least, *VehicleSpeedCheck* block monitors the vehicle speeed signal coming as input. We need to take into account conditions when the vehicle speed is not available for some reasons or there is a fail on it. A possible fail is captured by checking the vehicle speed fail status signal. Even here, we use a timer to decide whether to actually consider the fail on vehicle speed or not; fail status signal must be present for 60 seconds to decide that there is a real fail or we must get RTE error for VehicleSpeed. If so, the ExternalTemperature value is set to zero to signal the situation. For the BSW this will be considered a fail condition and the user will be informed about it.

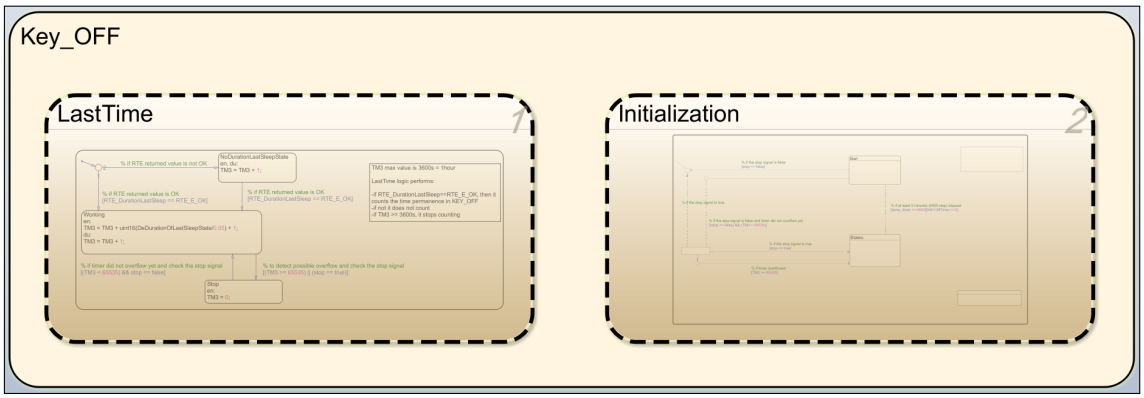

Figure 4.7: Key OFF blocks

Key OFF state

Key OFF state contains functionalities that work only when the car ignition switch is moved to the second position and/or is maintained in that position. Main block are reported in figure 4.7.

Inner blocks *LastTime* and *Initializaion* are independent each other.

The purpose of Key Off state is to set the ExternalTemperature output at Key Off in relation to certain conditions. *LastTime* block is in charge of counting the time permanence at Key Off and a local variable for it is always updated. This value is used by the *Initialization* state which, on its turn, depending on the time permanence, evolves its internal finite state machine. As the VF specification prescribes, during the first five minutes the ExternalTemperature output is not updated, then it changes according to the information provided by the LastTime block. If a battery reset has happened (default transition) or we are coming from a fail condition on the vehicle speed or we move from a condition where the temperature sensor was failed, the ExternalTemperature is set to the current measured temperature value, otherwise it is set to the value kept in memory that is nothing but the last value sent as output. Every minute the time permanence in Key Off is checked and if it is

greater than a certain time threshold, then ExternalTemperature output is updated according to the following equation:

$$ExternalTemperature = \frac{temp_{value} * (TM_3 - Thr_1) + TMemory * (Thr_2 - TM_3)}{(Thr_2 - Thr_1)} \tag{4.1}$$

where:

- $temp_{value}$: current arithmetically averaged temperature value
- TM_3: time permanence at Key Off
- $TMemory$: last temperature value sent to ExternalTemperature output
- Thr_1: lower time threshold
- Thr_2: higher time threshold

Basically, the above equation computes the weighted average that represents the updated ExternalTemperature output. Thr_1 and Thr_2 are constant values, therefore when the time permanence TM_3 becomes bigger, the value of Thr_2 - TM_3 decreases, while the result of TM_3 - Thr_1 increases and the weight on $temp_{value}$ gets heavier, therefore the actual value will tend to be the current arithmetically averaged temperature value, otherwise it will be closer to the last temperature value which is $TMemory$. The weighted average equation has been implemented by a *Function* in Stateflow.

When the time permanence is greater than *30 minutes*, Last ExternalTemperature is updated to the current averaged temperature value. Then, every minute the logic continues updating the output until the maximum allowed time is reached. In fact, LastTime block uses a *16 bit*-variable counter to count every step of *50ms* each and stops when *65535* steps are reached, which correspond to about *3276 seconds = 54.6 minutes*. The update logic at Key Off also stops its execution.

Diagnosis state

Diagnosis state includes functionalities that are necessary to know whether:

- There is a fail on the temperature sensor or not
- The measured value is outside the allowed range
- The information is not available due to a disconnection of the sensor itself from the system

For what concerns the knowledge of fails, another port is exploited. *TempSensorFail* port, that is a *DataReceivePort* (see table 4.1), gives the information about a possible fail that may occur when trying to deliver the temperature value coming from the sensor (*TempSensor* port).

In order to be sure that there is actually a real fail on it, a debounce operation on the TempSensorFail must be performed. In particular:

- After three consecutive cycle with the signal carrying a *fail* status, TempSensorFail is set to Fail_present
- After three consecutive cycle with the signal carrying a *no-fail* status, TempSensorFail is set to Fail_not_present.

Basically, the process needs to wait for three consecutive cycle that is $150ms$ (since the runnable entity is cyclically executed every $50ms$), before setting the actual fail status in both cases.

Sometimes the read temperature value may also be outside the sensor specifications, therefore a check on it must be performed. Temperature lower and upper limits are *-54.5 ℃* and *-99.5 ℃* respectively.

Finally, to get the presence or absence of the temperature device, the proper *TempSensorPresent* port that is a *ClientPort* (see table 4.2), is used by calling the get operation defined in the VF specifications.

All these checks work in Full Communication only because require the use of the CAN Network to update the ExternalTemperature output and both at Key ON and Key OFF.

4.2.5 Model tests

As already stressed before (see chapter 3), testing the model is really important to check whether errors in the design are present and to deliver a perfectly working product to the customer. Practically speaking, testing means creating test cases to stimulate the system model and analyse the results. Test cases must be compliant with the input specifications which can be read from the requirements document. Produced results have to be compared with the expected outputs. A picture, summarizing the concept, is shown in figure 4.8.

To accomplish this essential task, once again Simulink has been exploited since it makes at our disposal a way of generating custom waveforms according to the needs. The used Simulink block is called *Signal Builder*. Signals for every input and for the expected ExternalTemperature output have been generated.

A *ModelInput* block contains the Signal Builder and provides signals as test cases to the system model for its eight ports, while a *ModelOutput* block is placed at the output to capture the ExternalTemperature value. All the signals are created considering a $50ms$ time step, since it is the time period at which the model will be cyclically evaluated.

As we needed to compare the output signal, the expected ExternalTEmperature value has been generated inside the Signal Builder.

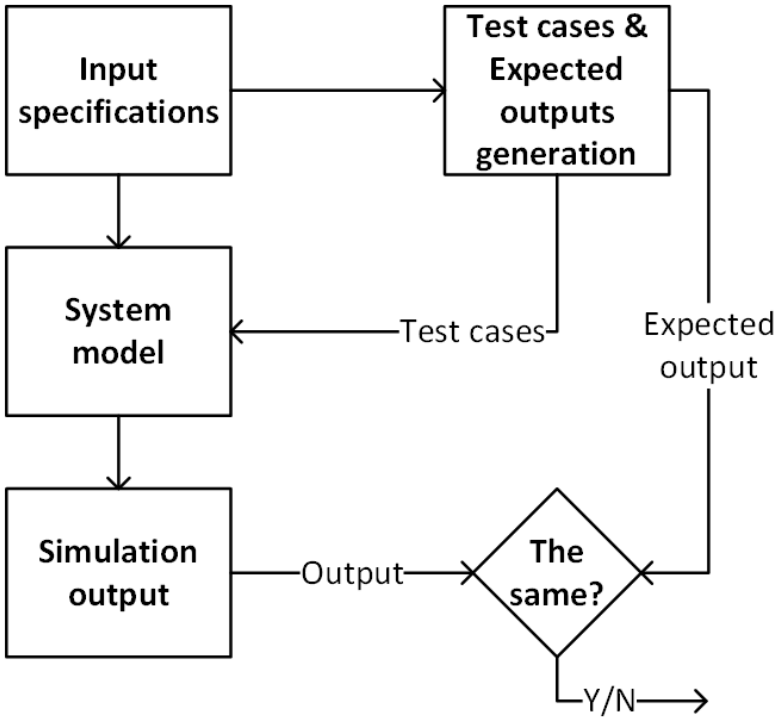

Figure 4.8: General model testing process

Finally, to actually be able to compare such quantities, an *Assertion System* block has been included in the model, just for testing purpose. It is in charge of doing a compare and allows to have a tolerance of $150ms$ in case of inequality between the model output and the expected one. If the inequality persists longer than the tolerance, the simulation is automatically stopped and provided with an alert.

After having integrated *TargetLink* tool, the model looked like the one shown in figure 4.9. It contains *ModelInput* block, *TargetLink Subsystem* block including the External Temperature Management (ETM) Chart (see fig. 4.2 for the chart), *ModelOutput* block and *Assertion System* block. Furthermore, other required blocks have been included: *TargetLink Main Dialog* and *MIL Handler*. The first block allows to open the settings panel to set the options for the simulation, code generation and others, while the second block enables some features, like Overflow checking, for MIL simulation.

For the sake of an example, input signals (see section 4.2.3) can be customized as in figure 4.10.

In this particular case, I wanted to test the model behaviour when the CAN network moves from being not-active to active (*No Communication* to *Full Communication*).

As already mentioned before, from figure 4.10 there are four more signals (the bottommost) with respect to the input ports the TargetLink subsystem (figure 4.9) should have. At this phase we want to also test the response coming from the lower level (Basic Software (BSW)) through Runtime Environment (RTE) calls, when getting data through *DataReceivePorts*. Not all the signals have this possibility, but in our External Temperature Management (ETM) case they are: *TempSensor*, *CarIgnitionSwitch*, *VehicleSpeed* and *DurationLastSleep*.

Each and every signal is generated (see figure 4.10), for testing purpose, as follows:

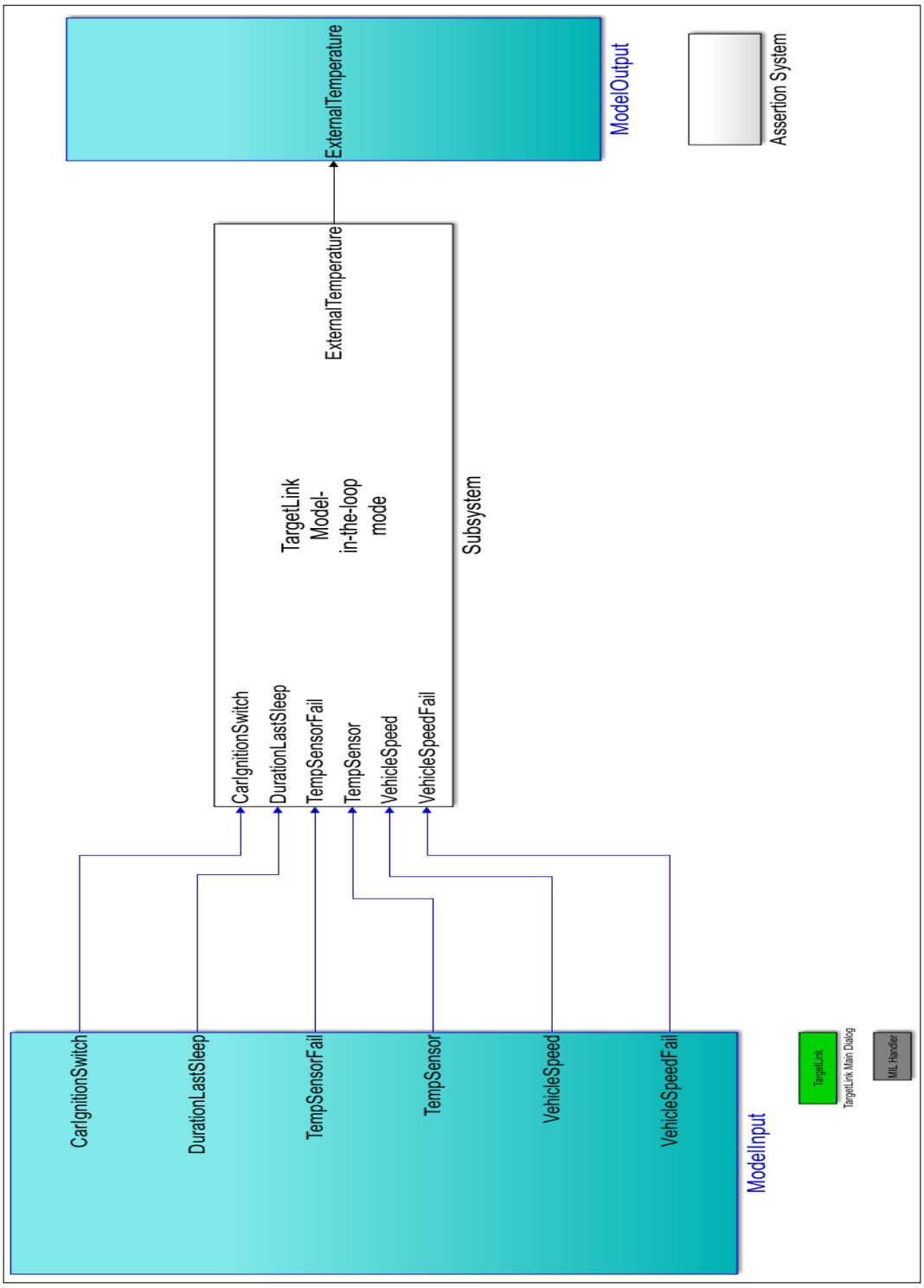

Figure 4.9: Model after TargetLink integration

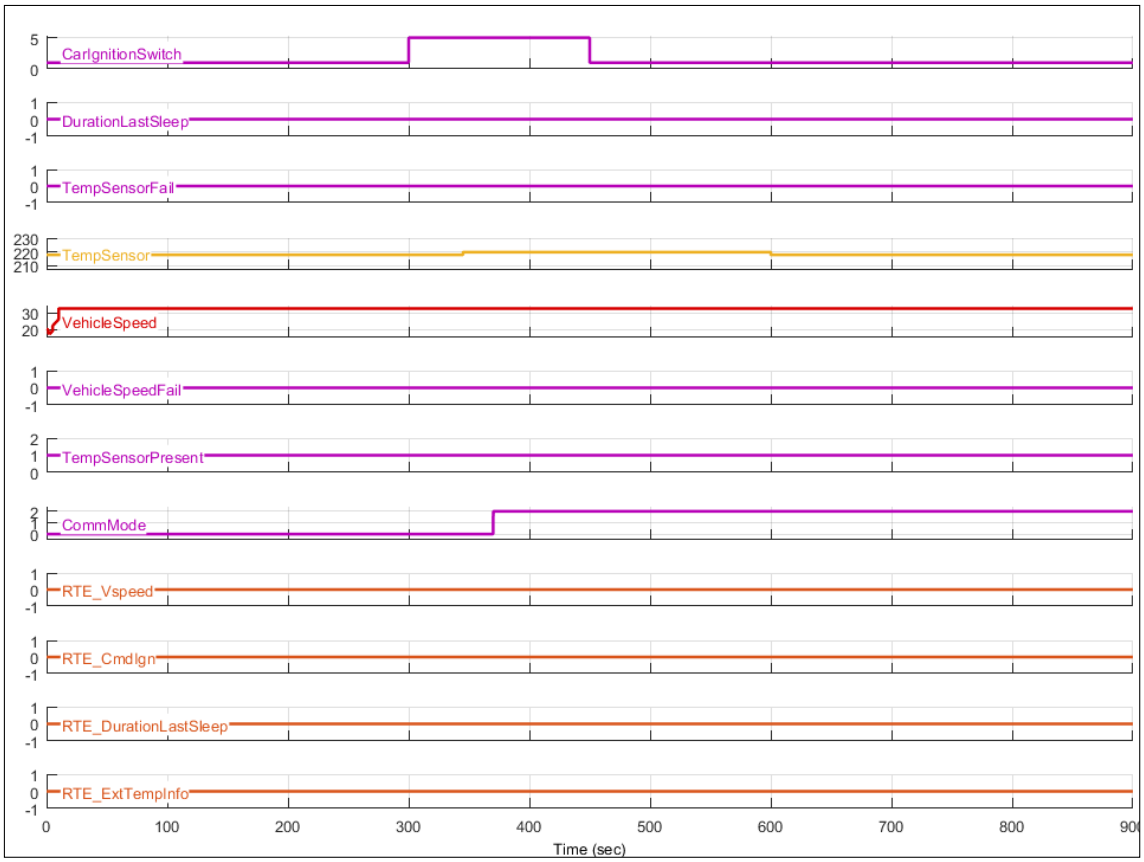

Figure 4.10: Input signals created with *Signal Builder* in *No Communication* to *Full Communication* test

- *CarIgnitionSwitch*: car ignition switch changes from being Key Off to Key On and then back to Key Off
- *durationLastSleep*: duration of the last sleep state is zero
- *TempSensorFail*: no fail is present on the NTC sensor
- *TempSensor*: temperature value coming from the NTC sensor. It has a little variation at some point in time
- *VehicleSpeed*: vehicle speed value is lower than the threshold (*30 Km/h*) just at the beginning, then it increases overcoming it
- *VehicleSpeedFail*: no fail is present on vehicle speed
- *TempSensorPresent*: NTC sensor is present
- *CommMode*: Communication mode, so CAN Network changes at some point in time from being inactive to active
- *RTE_Vspeed*: response from the BSW through RTE for vehicle speed gets no problem
- *RTE_CmdIgn*: response from the BSW through RTE for car ignition switch gets no problem
- *RTE_DurationLastSleep*: response from the BSW through RTE for duration of last sleep state gets no problem
- *RTE_ExtTempInfo*: response from the BSW through RTE for External Temperature info gets no problem

To accomplish the test of the model, the *ExpectedExternalTemperature* waveform has been also generated and it is reported in figure 4.11.

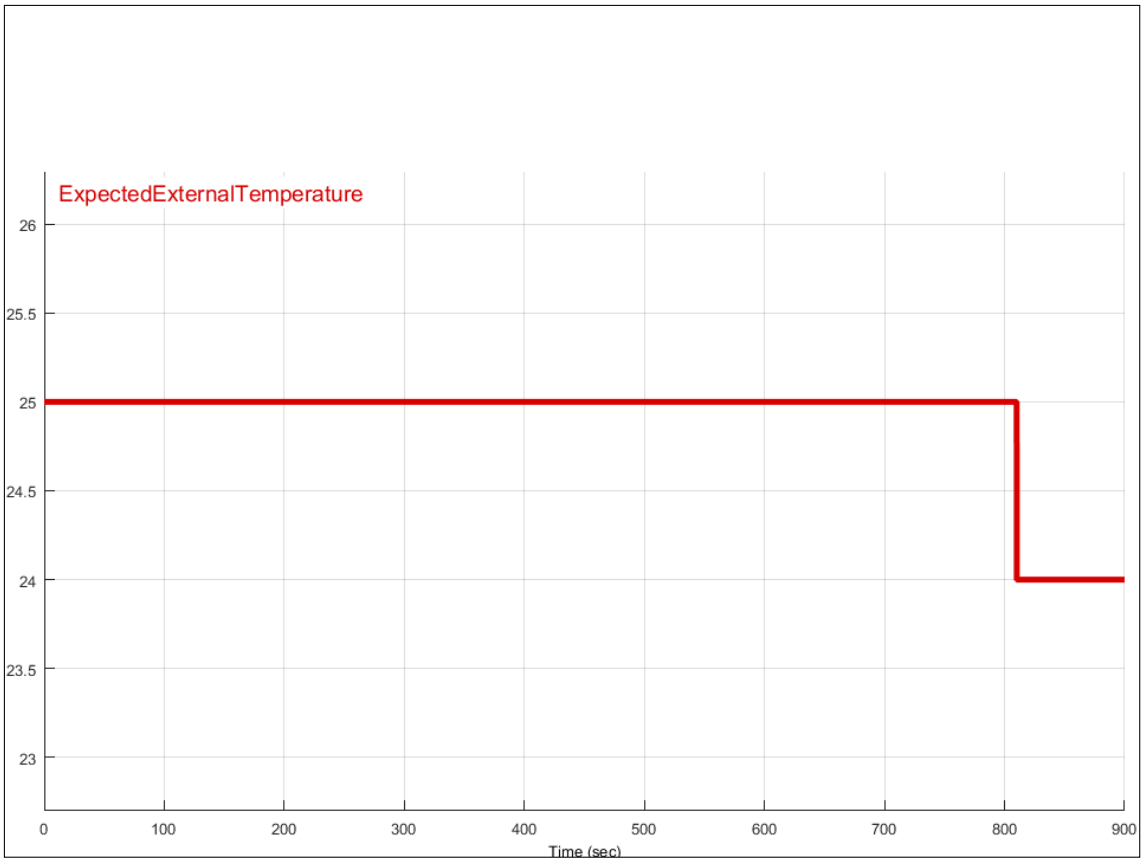

Figure 4.11: ExpectedExternalTemperature signal created with *Signal Builder*

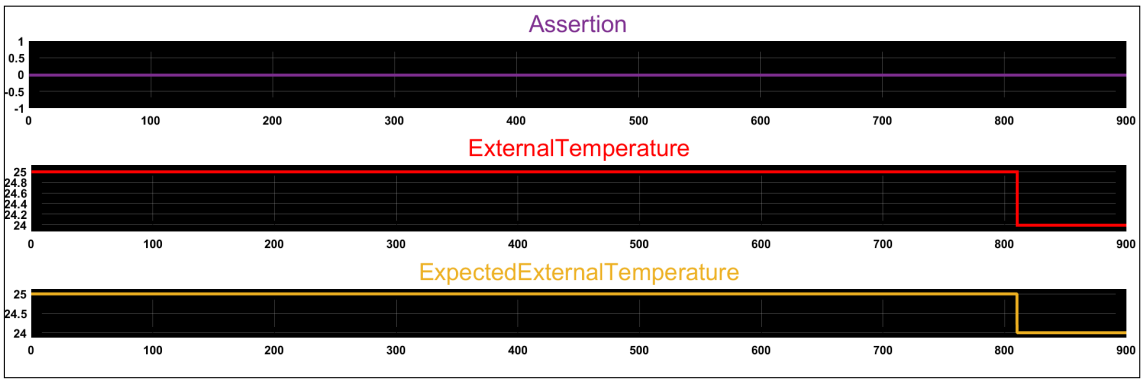

Figure 4.12: Assertion scope: result of the comparison between expected and actual output in *No Communication* to *Full Communication* test

Then, as mentioned before, we are able to compare the *ExpectedExternalTemperature* with the actual *ExternalTemperature* output (see table 4.3) of the main chart thanks to the Assertion block. The result of the comparison, for the test example, is shown in figure 4.12.

With those figures, I wanted to report an example of a test performed on the model implementation. Of course, many other tests have been accomplished for testing the most significant cases, namely most of the combinations of being in one state or in the other through the developed finite state machine, but they are not all reported here as they are many. Very important is the test of possible fails like on the temperature sensor input info and vehicle speed signal through its dedicated DataReceivePorts. Thus, it is worth showing the generated input signals with Signal Builder (figure 4.13) and the result (figure 4.14) of the Model-in-the-loop (MIL) validation when there is actually a fail on the temperature sensor (*TempSensorFail* signal).

From the provided figures, you can notice that the ExternalTemperature goes and remains at zero for some time almost in correspondence with the first change in the *TempSensorFail* signal from zero to one. This follows the specification of the algorithm for which the ExternalTemperature must go to zero when there is a fail

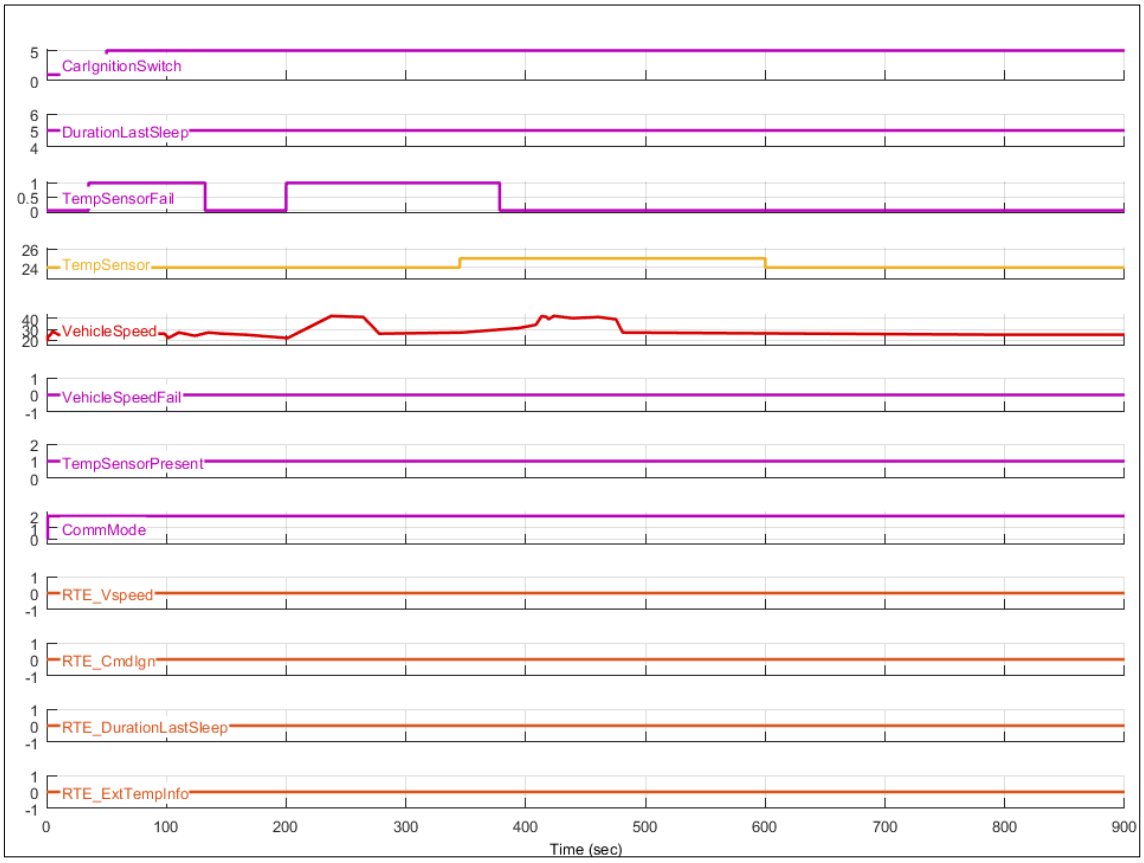

Figure 4.13: Input signals created with *Signal Builder* at *Full Communication* and *Key ON* with a temperature sensor fail

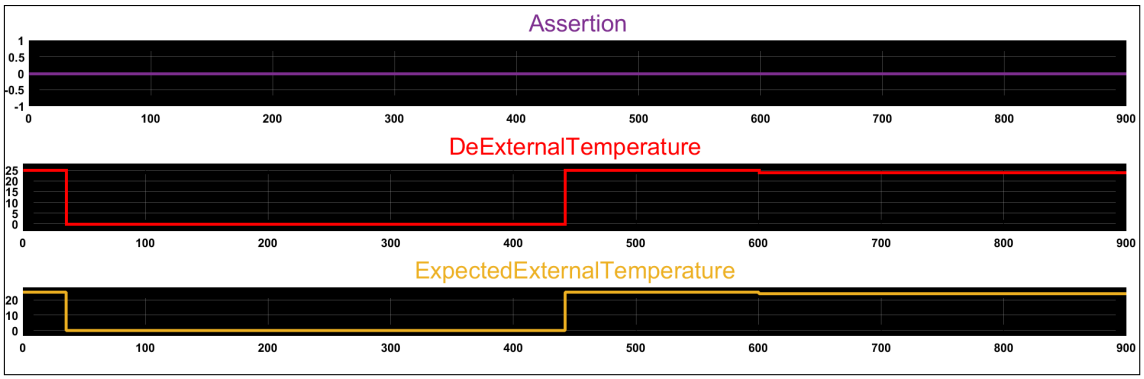

Figure 4.14: Assertion scope: result of the comparison between expected and actual output at *Full Communication* and *Key ON* with a temperature sensor fail

on such input information for at least three cycles. The event is captured by the *Diagnosis* state (see section 4.2.4.2 at *Diagnosis* state paragraph).

As a further remark, from figure 4.13 you can see that there are actually two time instants at which the TempSensorFail goes from zero to one, but as shown in figure 4.14, the ExternalTemperature output moves to zero only once. It is because at the point when the fail signal changes to zero (first time), the system is being in a *Not_Ventilated* state since the VehicleSpeed is lower than the predefined threshold. Due to that, the OverHeatingFilter block is not allowed to change the ExternalTemperature output because the measured temperature is greater than the current one (it is actually zero because of the sensor fail) (see section 4.2.4.2 and *OverHeatingFilter* state description in *Key ON* state paragraph).

Last but not least, the test of the system when there is a fail on the vehicle speed input (*VehicleSpeedFail* signal) has been carried out. The generated input signals with Signal Builder (figure 4.15) and the result (figure 4.16) of the MIL validation are provided.

Recalling the logic that manages the fail event on the vehicle speed signal (see section 4.2.4.2 and *VehicleSpeedCheck* description), from figure 4.15 you can notice that even if there is a fail at the very beginning, the ExternalTemperature output goes to zero only after about 120 seconds of simulation. This is due to the first Key Off condition (check *CarIgnitionSwitch* signal) at which a fail on the vehicle speed does not matter, therefore the time to wait to read the fail increases. It is realized by the system after some time at Key On that is required, as the specification prescribes, to finally consider the fail on the vehicle speed as true. The same behaviour is clearly visible again when the vehicle speed fail changes at one later in time. It is worth highlighting that even though the fail moves to zero in between, the ExternalTemperature does not update because of the logic inside the *OverHeatingFilter* state (like the previous test example).

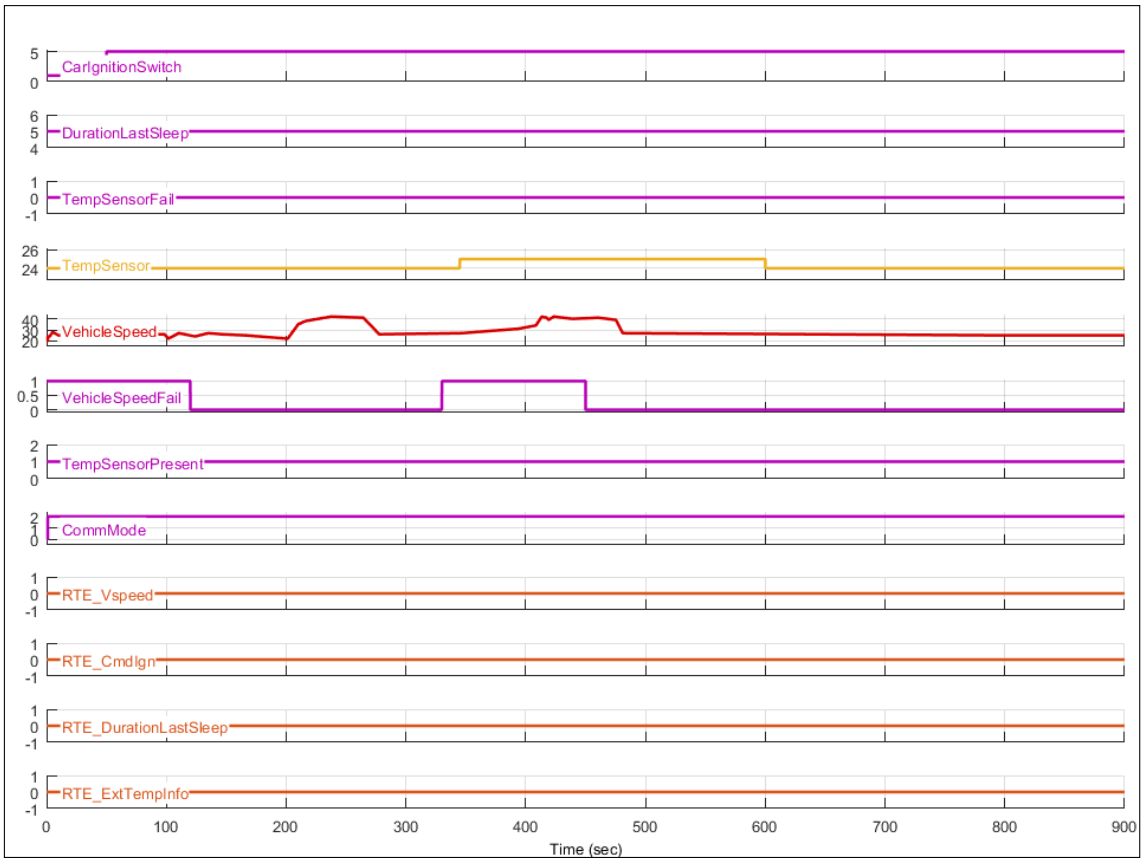

Figure 4.15: Input signals created with *Signal Builder* at *Full Communication* and *Key ON* with a vehicle speed fail

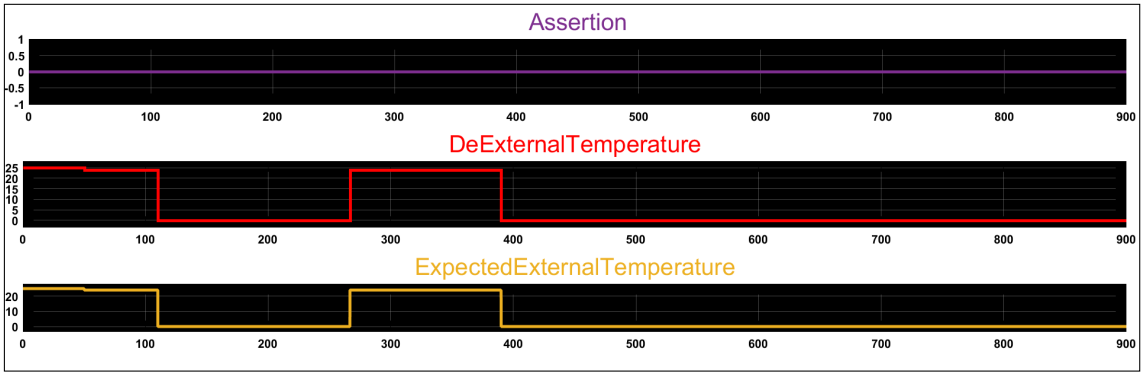

Figure 4.16: Assertion scope: result of the comparison between expected and actual output at *Full Communication* and *Key ON* with a vehicle speed fail

In the next section, the code generation and integration steps will be described.

4.2.6 VF code generation and integration

The goal is to deploy the implemented software on a real Body Computer Module (BCM) and the code generation phase is carried out after the algorithm has been validated in the simulation environment.

TargetLink (by dSpace) has been integrated into MatLab (supporting the AUTOSAR standard) at the Model-in-the-loop (MIL) phase and has been also used to auto-generate the code for the specific hardware platform - in my case, the target BCM mounts a 32-bit microcontroller. The C code generation options can range from ANSI C code to optimized fixed- or floating-point code for AUTOSAR platforms and MISRA C compliance requirement. TargetLink tool guarantees an efficient C code as it would have been written by hand. To achieve that, in terms of code occupation and CPU execution time, TargetLink relies on optimization techniques to manage the complexity and save execution time, and ROM and Stack size. It also offers a *Data Dictionary* to handle variables, data structures, tasks and functions of the project. The proper settings on TargetLink have been set, in order to generate an AUTOSAR-compliant and Generic ANSI-C code (see figure 4.17).

Before actually generating the code, some parameters have been set. TargetLink provides a *Propety Manager* to configure every SW-C input and output but also constant and local variables. The suitable data and class types must be set in order to succeed in the code integration.

Then, by clicking to the *Generate Code* button in the TargetLink Main Dialog, the code generation process starts. Once finished, if no errors occurred, the developed vehicle function consists of an AUTOSAR-compliant ANSI-C code - *TLProj* folder is created. It basically contains two files: *ETM.h* and *ETM.c*.

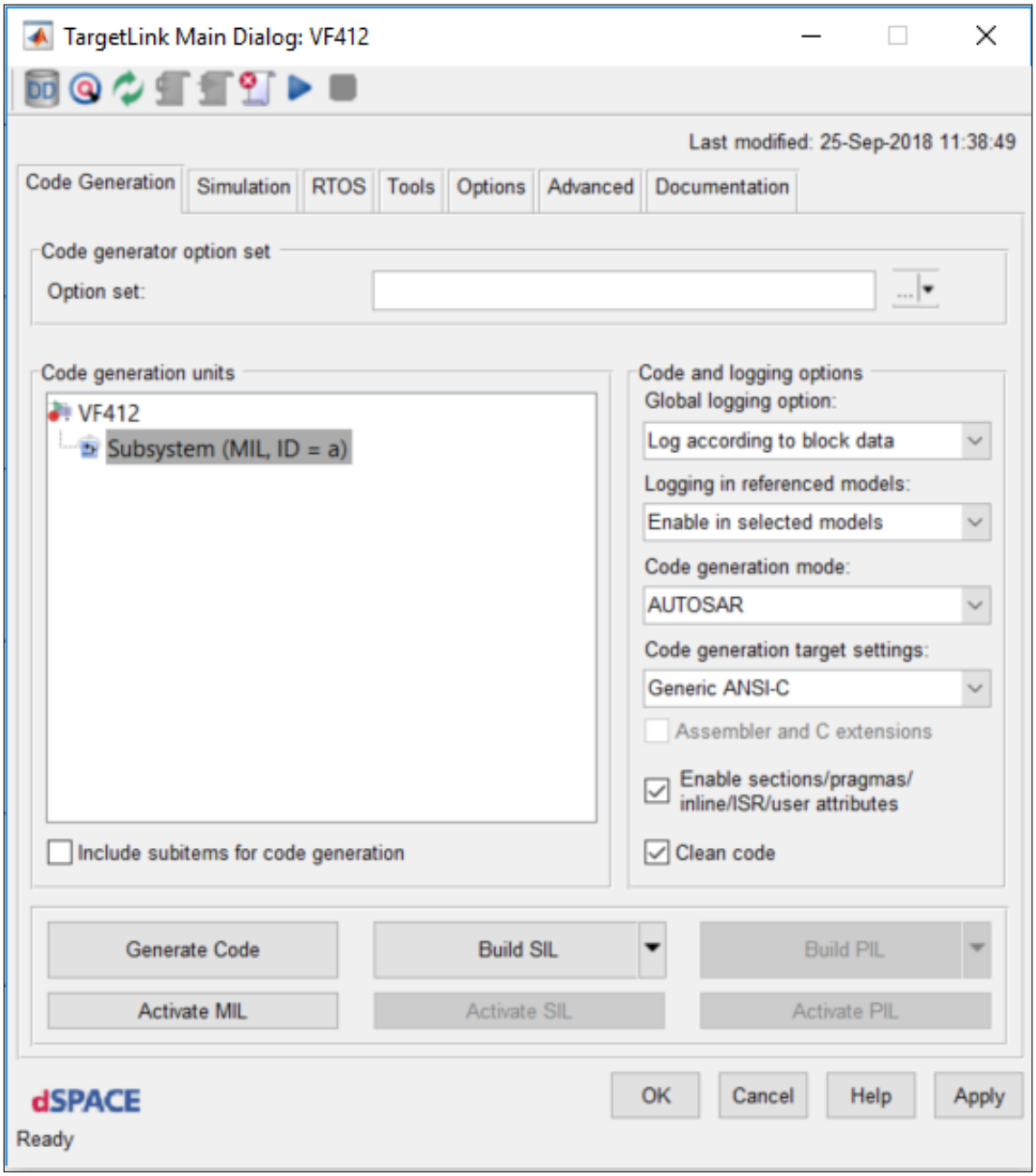

Figure 4.17: TargetLink Main Dialog settings

The header file declares the runnable entity as a C-function and other required elements, while the source file defines the functionality of the runnable entity for the ETM vehicle function. Moreover, documentations of the generated code are automatically produced.

Next step in the development flow is the integration with the BSW, as it provides all the required services and functionalities, like the Operating System (OS), that allow to access through RTE calls to the drivers for managing registers of microcontroller and needed peripherals. This concept follows the AUTOSAR standard (see chapter 1), for which from the application space is not possible to directly point to the hardware resources.

Recalling the concept of the standard, there are three main layers: *Application layer*, *RTE layer* and *BSW layer*. Hence, code integration has been accomplished by combining the Application layer with the lower levels and to achieve it the *Softune Workbench* has been used.

Another fundamental part of the whole software is played by the *SensorIn/SensorOut* components. Relocating them among ECUs is not always possible because of the dependency with the hardware. They actually are AUTOSAR Atomic Software Components, used for sensor evaluation and actuator control, therefore they belong to the Application layer even though hardware-specific. In the case of ETM, SensorIn component provides data to ETM component's input, in particular - last sleep state duration and external temperature value together with its fail status. SensorOut component provides instead the external temperature value as output from the BCM after being properly translated from voltage measurement to Celsius degrees.

Lower levels implementation, for microcontroller and peripherals interaction, has been developed by other teams and then all the needed files have been incorporated in the same project to be cross-compiled for the target hardware.

Hereinafter some additional details about the BSW layer are provided in order to understand how the developed software will be actually executed on a real BCM in co-existence of other functionalities.

4.2.6.1 OSEK OS standard

The BSW encompasses the OS, which must follow the *OSEK* standard. OSEK standard was founded in 1993 by a German automotive company consortium (*BMW, Robert Bosch GmbH, DaimlerChrysler, Opel, Siemens, and Volkswagen Group*). In 1994, the French cars manufacturers *Renault* and *PSA Peugeot Citroen* joined the consortium as well. It basically specifies interfaces to multitasking functions and generic I/O and peripheral access, therefore it is architecture-dependent [21]. Focusing on tasks, they are executing processes that are executed according to their timing requirements and scheduled by the OS Scheduler.

OSEK standard differentiates two task types with different behaviours: *Basic task* and *Extended task*.

- **Basic task**: never blocks therefore it runs till completion
- **Enhanced task**: can go to sleep and also block.

Figure 4.18 shows the states that a basic task can achieve and its transitions towards them.

A basic task must stay in the Suspended state whenever it has terminated (completion of its job) or it has not been activated yet. Before Running, it must move to the Ready state but it can be also preempted (higher priority task).

Figure 4.19 shows instead the states that an extended task can achieve. It has an additional state, Waiting state, with respect to a basic task.

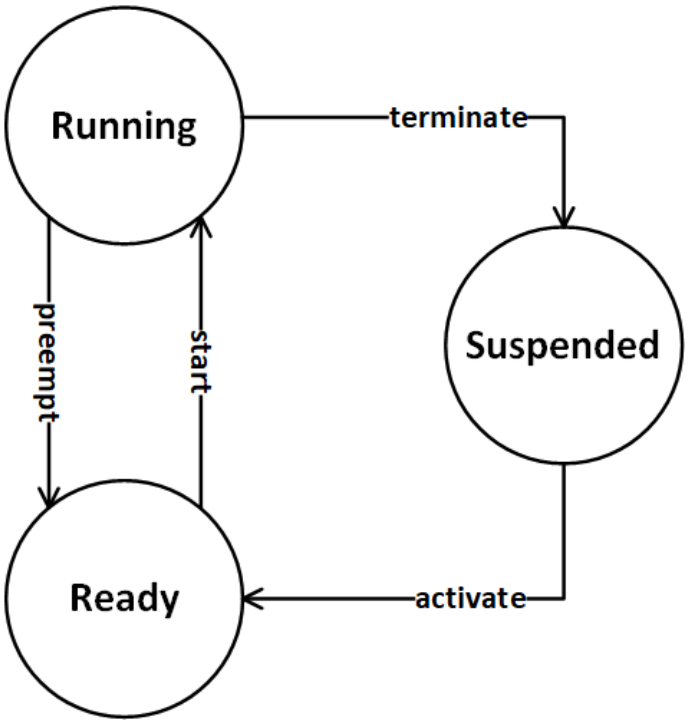

Figure 4.18: Basic task states and its transitions

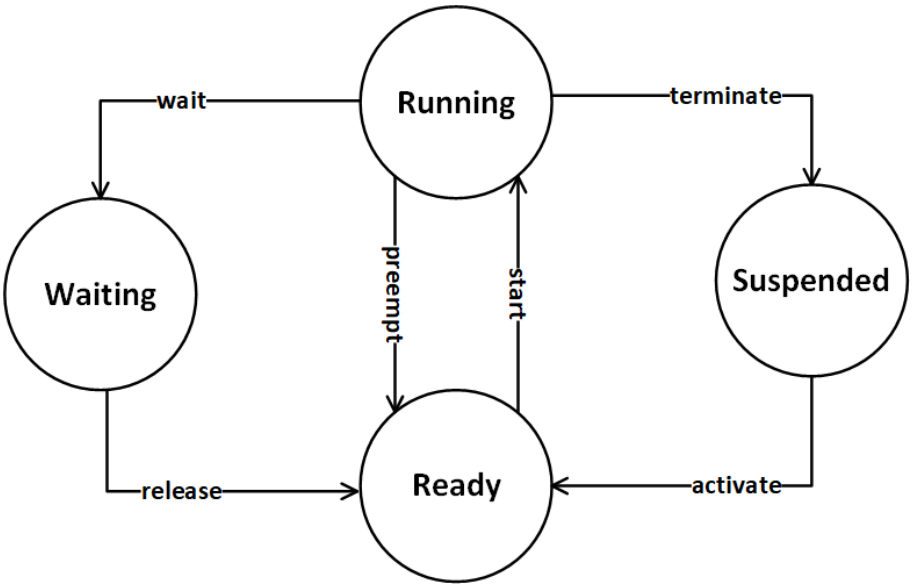

Figure 4.19: Extended task states and its transitions

The main difference is that an enhanced tasks can be blocked depending on event objects. An event can be raised by other tasks (either basic or enhanced) or interrupt routines. When task needs to wait for something, it is moved to the Waiting state instead of being suspended until event notification, which causes the task to be released and moved to the Ready state. The scheduling of tasks is managed by the OS Scheduler in two ways: *Preemptive scheduling* and *Non-preemptive scheduling.*

- **Preemptive scheduling**: the scheduler can block (preempt) a task in favour of higher priority tasks
- **Non-preemptive scheduling**: the scheduler can preempt a task in prefixed compile-time points only.

These two scheduling methods can be also mixed. Hence, tasks are scheduled according to the scheduling policy, but also based on their priority. The latter is statically assigned to every task by the user at design time. Priorities are labelled by numbers, the higher the number, the higher the priority of the task.

How is a software component executed?

A Software Component (SW-C) needs access to resources like memory and CPU, therefore the Runtime Environment (RTE) must provide the environment for it. The implementation, so the software itself, is invoked in response to Fixed-time schedules or Events, as already mentioned before. According to fixed-time schedule, the software may be run cyclically.

The actual implementation of a SW-C consists of a set of runnable entities. A runnable, for short, is nothing but a set of instructions (implemented by the software component) that can be started by the RTE. Generally, every runnable is associated with a task (it can even host more than one runnable) and scheduled by the OS Scheduler at run-time on the Electronic Control Unit (ECU). In this context, the

RTE is in charge of guaranteeing the invocation of runnables at correct time instants and providing the functions/data that the component needs.

Concerning our *ETM* vehicle function, its software implementation will be included into a runnable entity that will be executed cyclically every $50ms$, according to the specifications. Thus, there are no events that can block the running algorithm, which can execute till completion every time it is called by the OS Scheduler. Of course, the runnable entity for ETM will run in co-existence with other possible runnable entities perhaps within the same SW-C.

Chapter 5

Tests and results

Last validation step performed by engineers is called Hardware-in-the-loop (HIL) validation (see section 3.3). It allows to execute the generated code into the target hardware, stimulating it through a hardware simulator. During this phase, a further testing process is performed in order to guarantee the correct behaviour as expected. In fact, with the help of emulators, the code downloaded into the hardware can be debugged while checking the communication buses.

In case of misbehaviours, making changes back on the model is always possible but it can often introduce a delay in the delivery of the project and costs.

The final aim of this thesis work is to demonstrate that the behaviour of the simulated model is comparable with the one experienced on the real Body Computer Module (BCM). Thus, the results of the HIL validation have to be collected and compared to the results of the Model-in-the-loop (MIL) validation.

As already mentioned in previous chapters, the BCM receives input data from the Controller Area Network (CAN) bus which are used to update internal data and evolve the algorithm. Then, *Vector CANalyzer* has been exploited, on Personal Computer (PC), to analyse exchanged data with the CAN bus,.

Vector CANalyzer is a software program that allows engineers to figure out

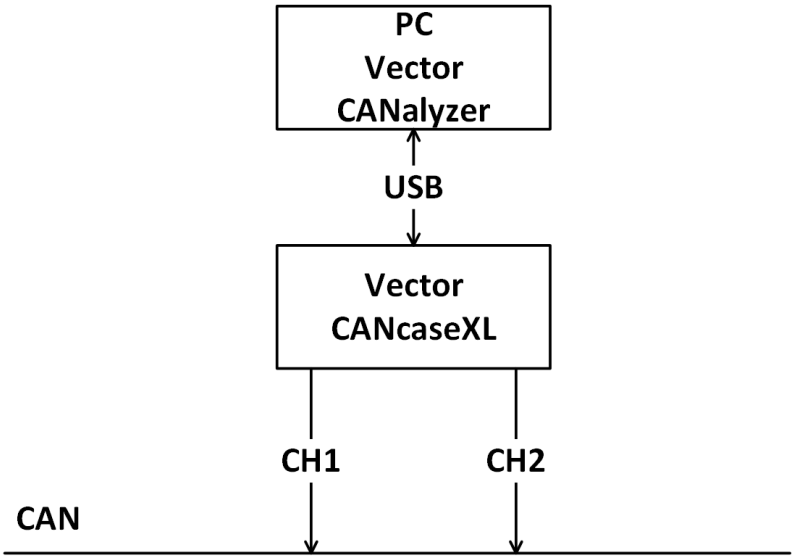

Figure 5.1: CAN tools: PC with Vector CANalyzer and CANcaseXL connection

whether the messages through the CAN bus are correct or not when testing their software implementations. It is important in order to deliver perfectly working algorithms. Many features are provided like, the tracing of the data traffic on the bus, the display of data messages, the statistical computations of the latter and others.

The target BCM, which executes the software algorithm after the download (see section 4.2.6), has been physically connected to a hardware simulator, which is needed to stimulate the BCM through its inputs and check the outputs. To achieve it, *CANcaseXL* has been also exploited. It is a Universal Serial Bus (USB) interface with two CAN controllers which allow to send and receive several CAN messages, such that a good test coverage can be obtained. Figure 5.1 shows how the connection between the PC with CANalyzer and CaNcaseXL is made.

CANalyzer tool, in conjunction with CANcaseXL, has provided me the possibility of seeing what was happening within the communication buses during the execution of the implemented and deployed algorithm on the BCM, and producing

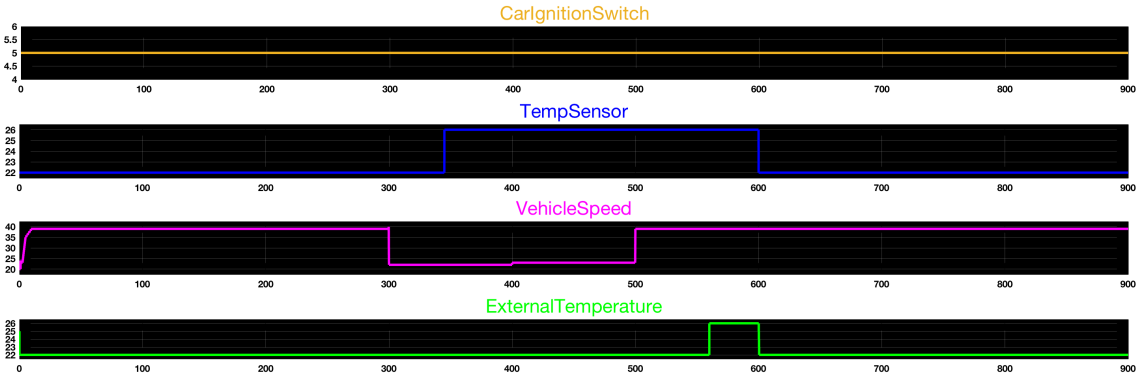

Figure 5.2: Normal behaviour at Key On: MIL simulation results with Matlab

results to be compared. In addition, the use of an emulator has made me capable of debugging the code to follow each and every executed instruction.

Model simulation results (similar tests to those in section 4.2.5 will be presented) have to be almost equal to the ones that come from the real-time validation. This permits to state that the code generation and integration processes have been successfully performed and system behaviour is still consistent with the expectations. Hence, in this chapter some significant test results are provided.

The normal case, in which there are no fails neither in the temperature sensor nor in the vehicle speed signal at Key On, is firstly provided. Figure 5.2 shows the main input signals involved and the ExternalTemperature output - they have been obtained at Model-in-the-loop (MIL) phase.

CarIgnitionSwitch is at Key On till the end of the simulation. *TempSensor* changes its value from 22 to 25 degrees, while *VehicleSpeed* is greater than the threshold until some point (300 seconds), then it decreases. Finally, it gets again greater than the threshold after about 500 seconds. *ExternalTemperature* value basically starts from its default value (25 degrees), then it goes to 24 as TempSensor input contains such value. When TempSensor reaches 25 degrees (at about 350 seconds), the value of ExternalTemperature does not change since the VehicleSpeed is

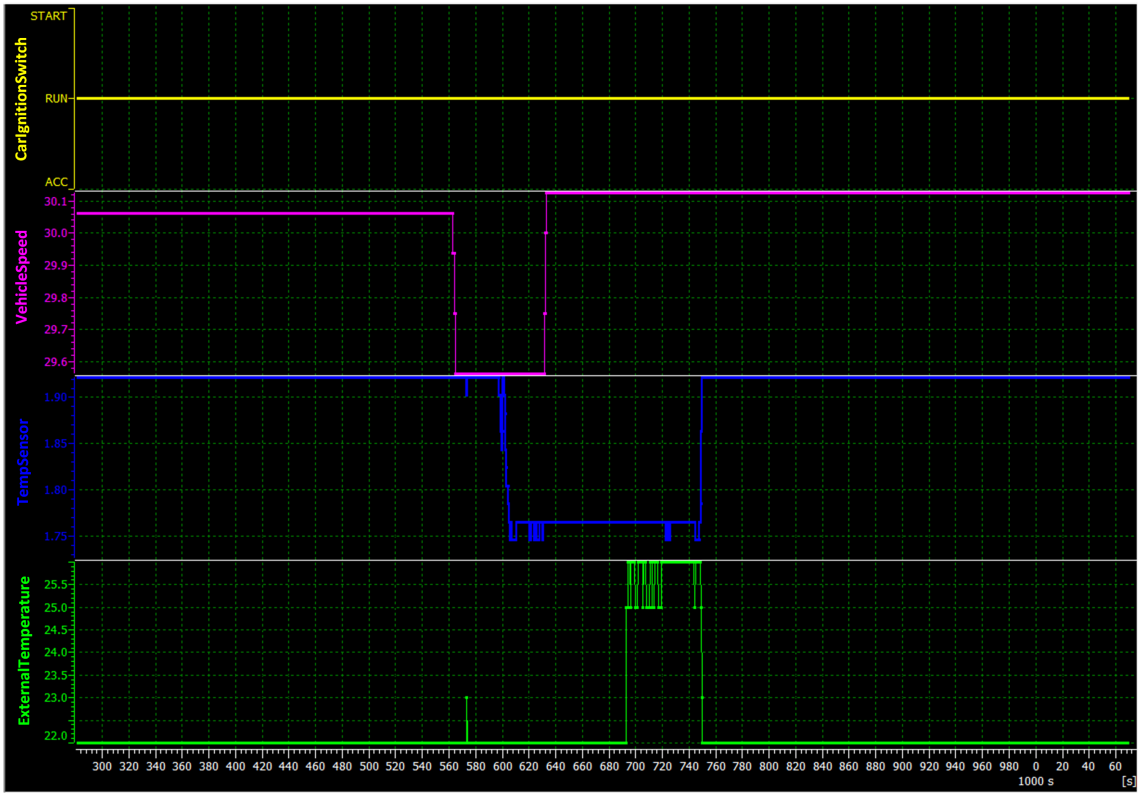

Figure 5.3: Normal behaviour at Key On: HIL validation results with Vector CANalyzer

lower than threshold (it implies the Not Ventilated condition for the system) and the new acquired temperature value is greater than the current ExternalTemperature, but it changes to 25 degrees after some point when the system goes into the Ventilate condition.

To compare and validate the behaviour, figure 5.3 reports the results analysed after deploying the code on the BCM and running the signals tracing with Vector CANalyzer.

It can be noticed that *CarIgnitionSwitch* is at Key ON (also called RUN) for all the time. *VehicleSpeed* is greater than the threshold till some point in time; during that time the *ExternalTemperature* value is updated according to the value coming from the *TempSensor* input. Once the VehicleSpeed decreases below the threshold

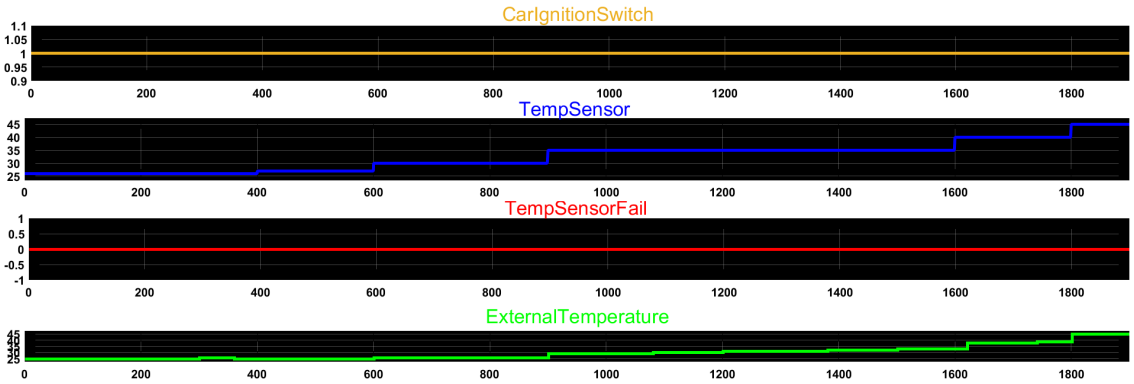

Figure 5.4: Normal behaviour at Key Off: MIL simulation results with Matlab

for enough time, then the system is considered as Not Ventilated - in fact even though the TempSensor value increases (the actual signal on the plot corresponds to the voltage read from the sensor and the lower the voltage, the higher the temperature (refer to 4.2.2 and Negative Temperature Coefficient (NTC) sensor description)), the ExternalTemperature does not change. The latter is updated only when the system moves to the Ventilate state after being the VehicleSpeed greater than the threshold for the predefined time. Finally the ExternalTemeperature value decreases as well as the TempSensor value.

After the comparison, we can state that the normal behaviour of the developed vehicle function is validated with respect to the model test (figure 5.2).

Another test has been performed at Key Off condition to check the behaviour of the logics in LastTime and Initialization independent states. The first state takes trace of the time permanence at Key Off, while the second state updates the ExternalTemperature output according to different conditions (refer to chapter 4.2.4.2).

Figure 5.4 shows the MIL validation results. Even this time, there are no fails.

Simulation time has been very high, in order to test the condition at which we

have a time permanence greater than *30 minutes*.

CarIgnitionSwitch is at Key Off for all the simulation time. *TempSensor* input varies many times during the simulation in order to make more updates on the output. In fact, ExternalTemperature output does not change until *300 seconds = 5 minutes*, as the specifications prescribes, then it is updated to *TMemory* that is the last ExternalTemperature (by default it is 25 since we are not coming from other key conditions). After one minute, the output is updated according to the weighted average formula (4.1). The latter is applied every minute as long as the time permanence is lower than *1800 seconds = 30 minutes*. Once that time has been reached, the algorithm moves to the last state where the ExternalTemperature output is updated to the current averaged temperature value. The output would continue to be updated until *65535 steps = 3276 seconds* which are *54.6 minutes*.

By running the *HIL* validation (see figure 5.5), we are able to check whether the behaviours are comparable each other.

Results are comparable, even though it has been difficult to provide the same *TempSensor* inputs of the MIL validation with the use of a physical potentiometer. It is just important to compare the logic from a functional point of view, without worrying about the actual temperature values. The states evolution is visible as the ExternalTemperature output does not change for the first *420 seconds* as: for 5 minutes it must not change, then it must be updated to the *TMemory* value (until 6 minutes) and then the weighted average must be computed (from 6 minutes on); it means that for *120 seconds* after *300 seconds* the temperature output remained the same because TMemory and weighted average were identical to the last External-Temperature value. As already mentioned before, the ExternalTemperature output would continue to be updated until *3276 seconds = 54.6 minutes*, but to reduce test time the validation has been stopped at about *1900 seconds = 31.6 minutes* that is greater than the minimum time (*30 minutes*) to reach the last state.

The fail events have been also simulated. For instance, figure 5.6 shows the

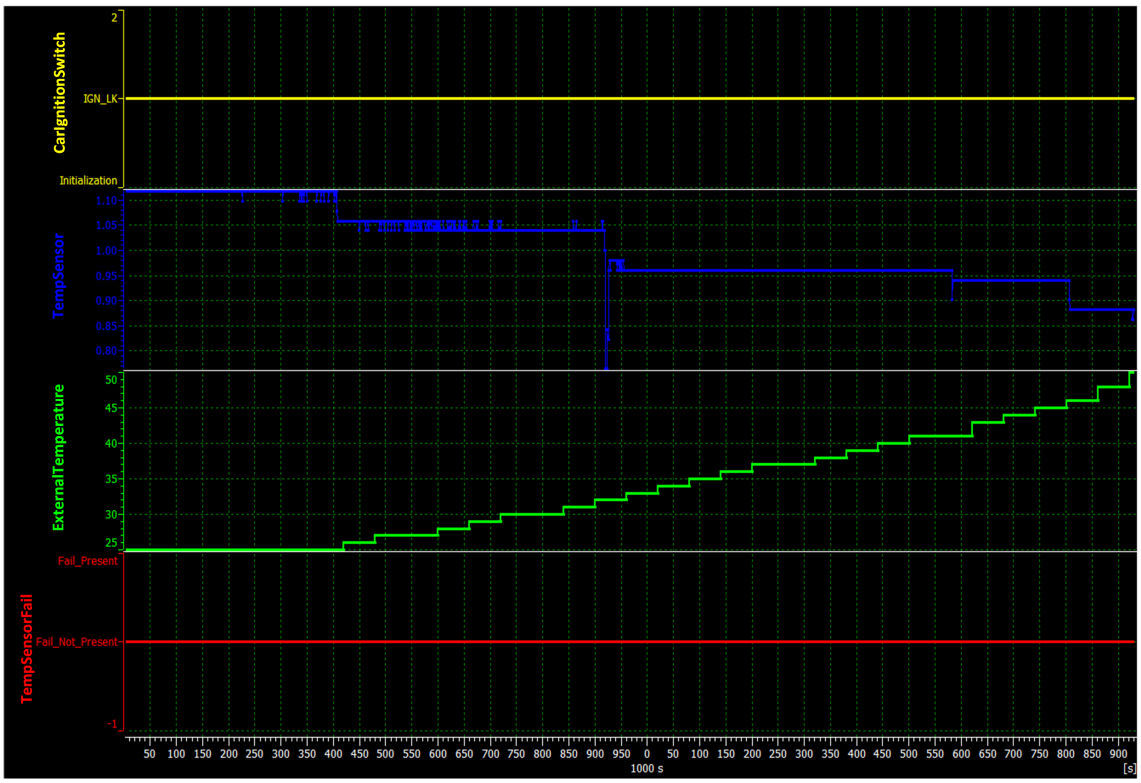

Figure 5.5: Normal behaviour at Key Off: HIL validation results with Vector CANalyzer

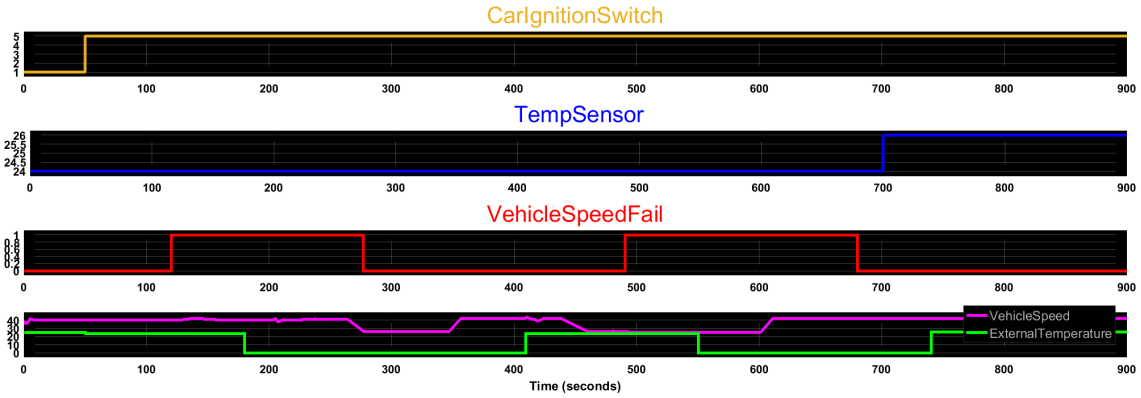

Figure 5.6: VehicleSpeed fail: MIL simulation results with Matlab

results obtained during MIL phase when there is a fail on the VehicleSpeed.

CarIgnitionSwitch moves from Key Off to Key On and remains stable till the end of the simulation. *TempSensor* is constantly at 24 degrees except for the last 200 seconds when it changes to 25 degrees. *VehicleSpeedFail* signal variates from zero to one more than once. *VehicleSpeed* value is greater than the threshold at the beginning and remains the same for some time, then it decreases but also increases afterwards - actually it does twice the same. Finally, *ExternalTemperature* follows the TempSensor value until the VehicleSpeedFail is at zero, then it moves to zero when a fail is detected (according to the logic, at least some time has to pass in order to actually consider the vehicle speed fail as true). In the meanwhile, the VehicleSpeed value decreases below the threshold, therefore even if the VehicleSpeedFail is restored to zero, the output value does not change, thus it remains at zero. The ExternalTemperature output can be updated to the TempSensor value only after the VehicleSpeed signal goes and remains greater than the threshold for at least the predefined time. The same behaviour is found later again in the plot.

Very similar results have been obtained during the validation phase - figure 5.7 reports them.

Once again, the TempSensor signal represents the input temperature value and

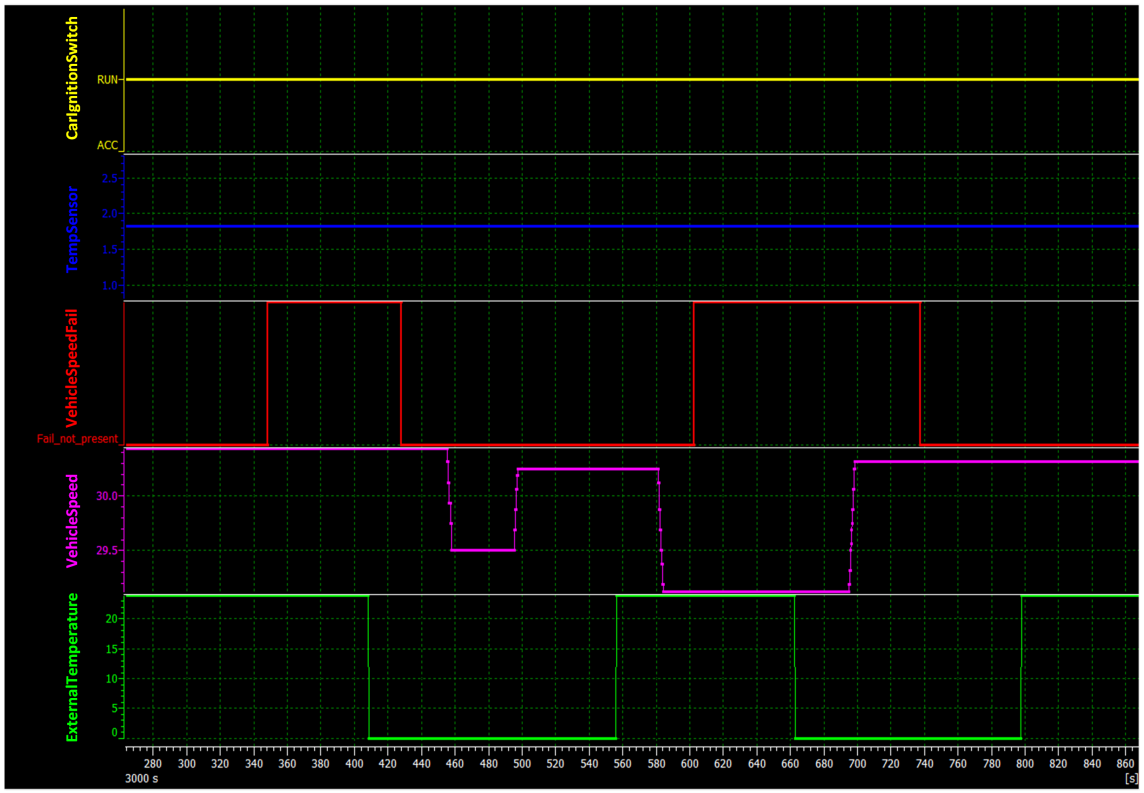

Figure 5.7: VehicleSpeed fail: HIL validation results with Vector CANalyzer

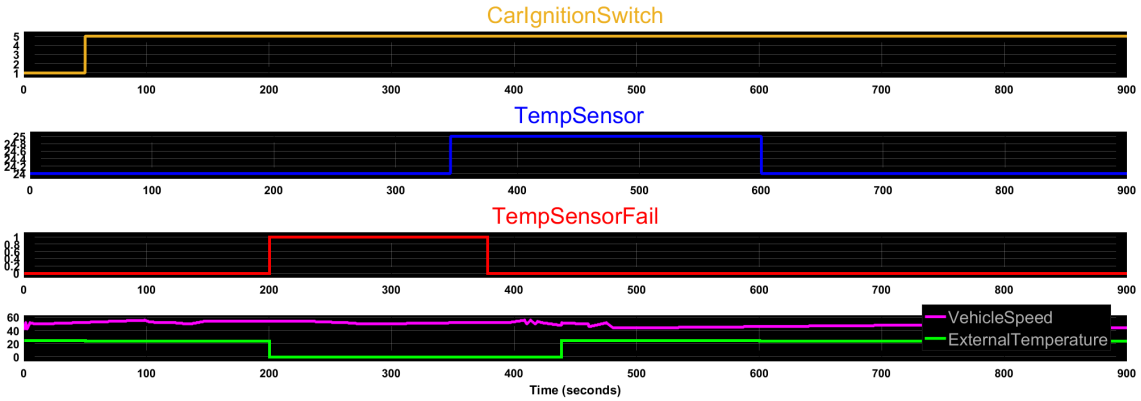

Figure 5.8: TempSensor fail: MIL simulation results with Matlab

it is captured as a voltage measurement. VehicleSpeedFail moves from Fail not present to Fail present more than once, while also the VehicleSpeed changes from being greater than the threshold and below the threshold. Of course, ExternalTemperature signal varies according to the conditions, already mentioned before and still comparable.

Last simulation consists on the case in which there is a fail on the temperature sensor. Figure 5.8 shows the results obtained during MIL phase when there is a fail on the TempSensor.

CarIgnitionSwitch is mainly all the time at Key On. *TempSensorFail* goes to one after 200 seconds and because of that the ExternalTemperature value goes to zero to signalize the fail - it is restored to the current TempSensor value only after 60 seconds after the fail has disappeared. *VehicleSpeed* is greater than the threshold for the entire simulation process.

The behaviour can be compared with the one reported in figure 5.9, where the validation results are shown.

To emulate a fail on TempSensor, therefore to switch TempSensorFail signal from Fail not present to Fail present, a physical jumper on the hardware simulator has

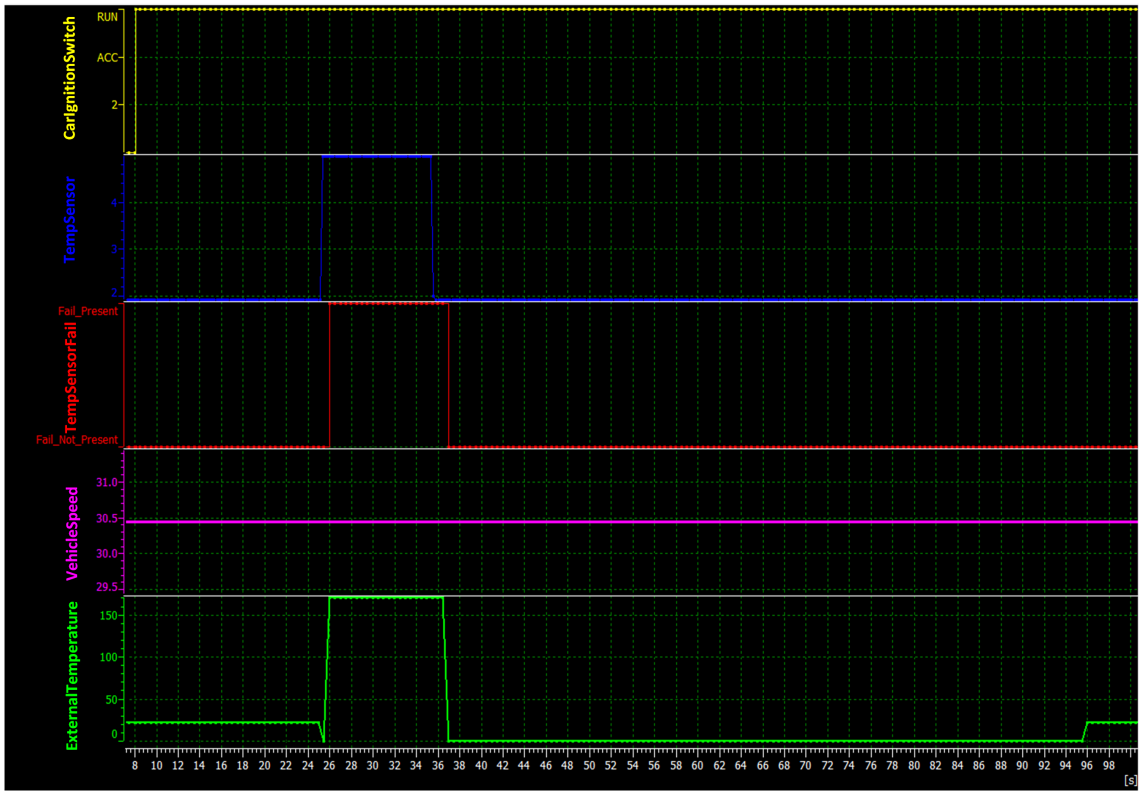

Figure 5.9: TempSensor fail: HIL validation results with Vector CANalyzer

been used. By removing it, the Open Circuit (OC) is simulated and TempSensorFail and TempSensor signals change accordingly. TempSensor is pulled up to 5V - see figure 5.9. As a result, the ExternalTemeperature output changes from being the current TempSensor value to the value that is used to notify a fail. At about 25 seconds, it can be noticed that its value goes to zero for a little time, then it is set to a high value (it is actually *170.5*) - the latter action is performed by the BSW, in fact the application only sets the value to zero for a little time. Finally, after about 60 seconds (as for the MIL case fig. 5.8), during which the VehicleSpeedFail signal is at Fail not present, the ExternalTemperature output returns to the previous value.

In this chapter, the most significant test results have been reported. However, to properly validate the developed algorithm many other tests have been performed, which allows to state that all the implemented functionalities fulfil the project specifications.

Conclusions

In the current technological age, a vehicle is provided with several features that require several ECUs as well. As a consequence, the software that implements such new functionalities becomes too complex and lines of code grow out of all proportion. This trend provides valued functionalities for the driver, but it must be controlled in some way in order to make the software perfectly satisfying the CPU and memory capabilities of the target hardware.

The goal of this thesis work was to provide a solution to the overheating effects that affect the temperature sensor in a car. It has been achieved by adopting Model-Based Software Design (MBSD), that is one of the most used methodology for automotive software development. In fact, MBSD allowed to manage the aforementioned complexity in the software and produce a bug-free C-code for production use. In addition, the software has been implemented to be AUTomotive Open System ARchitecture (AUTOSAR)-compliant.

As a first step, Original Equipment Manufacturer (OEM) vehicle function requirements have been analysed and studied from the provided document to understand the context, the logic and the algorithm to be implemented.

Since the beginning, the model has been conceived to be as much optimized as possible, therefore a draft model by hand has been produced in order to have an idea on what functionalities needed to develop and how to handle the optimization purpose. An optimized model will be translated into a lightweight *C-code* for embedded

systems deployment.

After some refinements, the implementation model has been moved to the *MatLab (Simulink/Stateflow)* environment. A finite state machine has been built that evolved through the states according to specific conditions. The next phase is called Model-in-the-loop (MIL) as improvements on the model can be applied while testing its behaviour from a functional point of view. Thus, test cases (specifications compliant) have been generated to validate the model against the OEM requirements. It is worth highlighting the fact that the developer, when designing the future software, must pay attention and worry about the model in development, such that he can get confident with his work earlier in the design phase.

Afterwards, the system model has been automatically translated into an optimized C-code, thanks to the use of the *TargetLink* tool, as a software component (application) to be integrated with the Basic Software (BSW) platform. Then, the whole package has been ready to be cross-compiled and deployed on a real Electronic Control Unit, a Body Computer Module (BCM) in this case.

Real-time validation tests, also called Hardware-in-the-loop (HIL), have been carried out to achieve a high degree of confidentiality with the developed software. For doing that, *Vector CANalyzer* tool has been used that provided a way of tracing CAN buses. Furthermore, *CANcaseXL* has permitted me to send and receive CAN messages, while debugging the code with an emulator.

To conclude, I would like to confirm that implementing a vehicle function from scratch has been a good learning experience. I have acquired the knowledge about automotive software development as I have worked on real project in an automotive company. Also, I have adopted a well-known software design methodology with the support of an automotive software standard. Moreover, I have learned new industrial software tools. Then, because the whole engineering development process, used in most of the automotive companies, has been followed, it has enabled me to obtain a

correct, optimized and requirements-compliant vehicle function which may be used for production.

List of Figures

List of Tables

Bibliography

[1] Flex Automotive. *CAN bus (Controller Area Network)*. http://www.flexautomotive.net/EMCFLEXBLOG/post/2015/09/08/can-bus-for-controller-area-network.

[2] AUTOSAR.org. *Application Interface*. https://www.autosar.org/standards/application-interface/.

[3] AUTOSAR.org. *AUTOSAR - Specification of the Virtual Functional Bus*. https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_SWS_VFB.pdf.

[4] AUTOSAR.org. *AUTOSAR Methodology*. https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_Methodology.pdf.

[5] AUTOSAR.org. *Software Component Template*. https://www.autosar.org/fileadmin/user_upload/standards/classic/2-0/AUTOSAR_SoftwareComponentTemplate.pdf.

[6] AUTOSAR.org. *Technical Overview*. https://www.autosar.org/fileadmin/user_upload/standards/classic/3-0/AUTOSAR_TechnicalOverview.pdf.

[7] Schätz B. et al. "Model-Based Development of Embedded Systems". In: *Bruel JM., Bellahsene Z. (eds) Advances in Object-Oriented Information Systems* (2002).

[8] Karsai G. "A Challenge and Opportunity for Model-Based Software Development". In: *Broy M., Krüger I.H., Meisinger M. (eds) Automotive Software – Connected Services in Mobile Networks* (2006).

[9] H.G. Gurbuz and Tekinerdogan B. "Model-based testing for software safety: a systematic mapping study". In: *Software Qual J* (2017).

[10] National Instruments. *Controller Area Network (CAN) Overview.* `http://www.ni.com/white-paper/2732/en/`.

[11] Texas Instruments. *Introduction to Controller Area Network (CAN).* `http://www.ti.com/lit/an/sloa101b/sloa101b.pdf`.

[12] Real-Time Systems Laboratory. *An introduction to AUTOSAR.* `https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf`.

[13] Silverio Martínez-Fernández et al. "A Survey on the Benefits and Drawbacks of AUTOSAR". In: *Conference: WASA'15: 2015 Workshop on Automotive Software Architecture* (May 2015).

[14] Mathworks.com. *Stateflow - User's Guide.* `https://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf`.

[15] Paul Green Motoyuki Akamatsu and Klaus Bengler. "Automotive Technology and Human Factors Research: Past, Present, and Future". In: *International Journal of Vehicular Technology, vol. 2013, Article ID 526180* (2013).

[16] Guido Sandmann and Richard Thompson. "Development of AUTOSAR Software Components within Model-Based Design". In: *SemanticScholar.com* (2008).

[17] STMicroelectronics. *LIN (Local Interconnect Network) Solutions, Application Note (AN1278).*

[18] Jnug Sung-Suk, Kim Jin-Ho, and Jeon Jea-Wook. "A Model-Based Design for Electronic Control Unit of Electric Motorcycle". In: *Electrical Engineering, vol 194. Springer, Berlin, Heidelberg* (2013).

[19] Vector.com. *Learning module FlexRay*. https://elearning.vector.com/vl_flexray_introduction_en.html?markierung=flexray.

[20] Wikipedia. *AUTOSAR*. https://en.wikipedia.org/wiki/AUTOSAR.

[21] Wikipedia. *OSEK*. https://en.wikipedia.org/wiki/OSEK.

[22] Justyna Zander, Ina Schieferdecker, and Pieter Mosterman. *Model-Based Testing for Embedded Systems*. Sept. 2011.

Acronyms

API Application Programming Interface. 16

AUTOSAR AUTomotive Open System ARchitecture. 6, 9–16, 18, 19, 21, 23, 24, 34, 42, 47, 73, 75, 89, 91

BC Branch Coverage. 39

BCM Body Computer Module. 7, 9, 10, 47, 49, 50, 73, 76, 80–82, 90

BM Bus Minus. 31

BP Bus Plus. 31

BSW Basic Software. 11, 12, 15–17, 19, 23, 44, 59, 67, 68, 75, 76, 87, 90

CAN Controller Area Network. 10, 20, 25–31, 50, 62, 64, 80, 91

CD Collision-Detection. 27

CPU Central Processing Unit. 9

CSMA Carrier-Sense Multiple-Access. 27

CSMA/CD Carrier-Sense Multiple-Access Collision-Detection. 27

ECU Electronic Control Unit. 6, 7, 11–13, 15–21, 23, 25, 33, 44–49, 78, 91

ETM External Temperature Management. 5, 10, 47, 50, 51, 64, 67, 75, 79

FCC Function Call Coverage. 39

FDMA Flexible Division Multiple Access. 31

HIL Hardware-in-the-loop. 6, 39, 45, 80, 90

HW Hardware. 15, 16, 20, 24

LIN Local Interconnection Network. 20, 25, 26, 29, 30

MAAB MatLab Automotive Advisory Board. 43

MBSD Model-Based Software Design. 5, 9, 10, 34, 36, 38, 42, 43, 47, 89

MC/DC Modified Condition/Decision Coverage. 39

MCAL Microcontroller Abstraction Layer. 17

MIL Model-in-the-loop. 6, 43, 64, 69, 71, 73, 80, 82, 84–86, 90, 92, 93

NTC Negative Temperature Coefficient. 8, 9, 84

OC Open Circuit. 87

OEM Original Equipment Manufacturer. 5, 6, 9, 48, 49, 89, 90

OS Operating System. 51, 75, 76, 78, 79

PC Personal Computer. 44, 80

PIL Processor-in-the-loop. 44

RT Real Time. 44

RTE Runtime Environment. 11, 15–18, 20, 21, 23, 54, 59, 67, 68, 75, 78, 79

SC Statement Coverage. 39

SDLC Software Development Life Cycle. 36

SIL Software-in-the-loop. 44

SW Software. 11–13, 19, 21, 24, 50

SW-C Software Component. 13, 23, 34, 73, 78, 79

TDMA Time Division Multiple Access. 31

USB Universal Serial Bus. 81

VF Vehicle Function. 9, 47, 48, 54

VFB Virtual Functional Bus. 13, 15, 18

VIL Vehicle-in-the-loop. 39, 45