POLITECNICO DI TORINO

Master Degree Course in Mechatronic Engineering

Master Degree Thesis

Software Development of the *Fuel Level Control* Vehicle Function for an Electronic Control Unit

Supervisor
Prof. Massimo Violante

Condidate
Marco Petrongari

Company supervisor
Eng. Mrs. Lorena Capuana

ACADEMIC YEAR 2017-2018

"Success consists of going from failure to failure without loss of enthusiasm"
-WINSTON CHURCHILL

to my family

Contents

Introduction

Since the production of the first Ford in 1908, the automotive industry has thrived all over the world and it has become one the bedrocks for the economy of the wealthiest countries. The use of a vehicle has quickly turned from an expensive object for the rich to an essential means of transport for everyone. Although the automobile identifies a great breakthrough and an engineering masterpiece, it is undeniable that, throughout the decades, it has undergone impressive change not only in terms of sales but also external features and inner functionality. Rather than a pure mechanical conveyance, today vehicles can be compared to high-tech devices where electronic systems guarantee a safer and more secure environment for the driver and millions of lines of code are run in a short period.

Within this context, the main goal of the thesis is to follow each single step behind both the software development of the *Fuel Level Control* vehicle function and its integration in an automotive electronic control unit (ECU). Overall, the whole project consists of five sections that intend to provide a top-down approach. In fact, starting from the description of the automotive environment, the body of the thesis brings out the software modelling phase without forgetting, in the final part, the importance of the test cases to attest the correctness of the implementation.

The first chapter highlights the difference between the software development process before and after the birth of AUTOSAR. In particular, this section explains the reasons that made car-makers and vendors think about a common standard and how this dramatic change had been achieved successfully. Therefore, the AUTOSAR bedrocks together with future objectives are analyzed in depth in order to illustrate the evolution of this consortium and the path that the automotive world endeavors to follow in the foreseeable future.

The AUTOSAR description and, in particular, the software architecture enable to focus on how a vehicle function is implemented. Consequently, the center of attention of the second chapter deals with the functionality description in terms of requirements, sensors and transfer functions. For this reason, understanding how the logic of the *Fuel Level Control* works, describing its mode of operation and studying all the interconnections are accurately covered in this part. Obviously, it aims to provide a high-level sketch where the most important transitions and states of the vehicle function are figured out.

The idea to follow a top-down analysis is particularly evident in the third chapter where requirements and objectives are matched with the *V-model* pur-

pose. As a consequence, what is clearly evident from this section is that the path to follow for the *Fuel Level Control* development is the result of a well-defined scheme rather than accidental events. In fact, the main goals are not only to link the first phases of the *V-model* scheme with the information provided before but also to justify the choice of the *Matlab/Simulink* environment for the model based design approach and the use of the *Stateflow* chart for the logic implementation. Thereby, while the output of the second chapter was a sketch on paper of the vehicle function, at the end of this section it turns into a proper software model.

The fourth chapter continues to discuss the remaining *V-Model* phases. In particular, it deals with the rising side of the diagram that takes into consideration each stage from the code generation to the final integration on the hardware target. In order to provide a clear idea of how to handle the code on a hardware device, all the required software that allow the code to be compiled and checked during a bench test are deeply discussed. Finally, the fifth chapter compares the outcomes of identical test cases that will be performed both in the simulation and bench tests with the aim of achieving marginal and expected differences that witness all the study conducted before.

Chapter 1

AUTOSAR

AUTOSAR

At the beginning of 2000, the consistent progress in the field of electronic systems and the ongoing importance of the software brought on revolutionary change in the automotive world. The need not only to integrate software applications and new components but also to exploit technological progress in favor of safer, more efficient and more secure automobiles provoked a great breakthrough in the vehicle software development.

Although in that period software development processes had already been realized by some automotive companies, the main drawback was that these were single development strategies. In addition to this, the collaboration with third parties would have only increased the complexity of the software without guaranteeing a long-lasting and reusable solution for future applications. In this scenario, the urgency to come up with a standardized outcome became a priority in order to lay the foundation for the future car development.

1.1 The idea of AUTOSAR

1.1.1 Difference with the past

In 2003, the worldwide partnership of automotive stakeholders and vendors founded AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecure): a standardized and open software architecture for automotive electronic control units (ECUs). Since then, this collaboration has been focusing on using software to control car applications and managing growing system complexity. In order to achieve these goals while keeping costs affordable, AUTOSAR has defined a set of specification for the description of software architecture, application interfaces, and methodology. Whereas, from the beginning the representatives of this partnership delineated a set of main aims to standardize out of competitiveness:

- Safety requirements;
- Redundancy activation;
- Scalability to different platforms variants;
- Implementation of basic functions;
- Transferability of functions from an ECU to another ECU;

- Integration of functional modules;
- Maintenability throughout the entire product life cycle;
- Software updates and upgrades;

However, the critical differences between previous and present software architecture were both the internal structure of an ECU and the way through which ECUs of two distinct sub-domains [1] communicate on the network (table 1.1). Before AUTOSAR, because of the absence of a common standard, both the ECU internal structure and the communication among ECUs developed accordingly to the past evolution of each sub-domain. Therefore, functional requirements to hardware and software were assigned to a one-on-one basis. This approach implied several negative consequences: first, any kind of change or improvement raised complexity and costs, secondly, this type of solution was difficult to re-use or integrate in other applications.

On the contrary, AUTOSAR aims to turn temporary into long-lasting solutions and to standardize the communication between the nodes[2] of the automotive network. By means of a layered structure, the ECU is essentially decomposed in three independent layers which make use of a dedicated virtual address space and interact each other through a specific interface. In this way, the level of abstraction proves to be the key factor that allows developers to overlook how to deal with the signal coming from the BUS but, simply, to handle the received data. Concerning the communication, AUTOSAR consortium has always preferred to take into consideration what has been already realized instead of inventing from scratch. For this reason, the easiest way to create a standard is to make a choice among current alternatives (LIN, CAN , etc.) or to set up a cooperation with standardization groups whose(Flex-Ray, MOST , etc.). It is undeniable that before making this decision, AUTOSAR members carried out a considerable number of research projects.

[1]Sub-Domain: nowadays, a modern vehicle network consists of about 50-70 ECUs. They are grouped in 5 different categories: Power Train, Vehicle Safety, Comfort, Infotainment, Telematics.

[2]Node: On a vehicle network, the term node stands for a single ECU.

Powertrain	It includes all systems related to the propulsion of the vehicle such as engine management and transmission control. The ECUs of this sub-domain are highly safety critical because a failure might make the driver lose control of the car.
Vehicle Safety	It provides safety assistance to the driver and includes systems that deals with the position and movement of the four wheels (anti-lock, braking systems, tire pressure monitoring, adaptive cruise control, airbag, and collision avoidance systems). Similarly to the Powertain domain, these ECUs are highly safety critical.
Comfort	It is mainly focused on driver assistance. For instance, electronic suspension, thermal management and parking assistance, air conditioning etc. are part of this set. Obviously, a failure might not affect the safety of the driver.
Infotainment	It includes control units associated to audio and video support in the automobile allowing information exchange between electronic system and the driver (digital broadcasting TV, audio streams, TFT displays, traffic and weather information systems).
Telematics	In comparison to the Infotainment sub-domain, Telematics takes into consideration systems that allow information exchange between the vehicle and the outside world (radio, navigation systems, payment, the internet).

Table 1.1: Sub-Domains in a modern vehicle network.

1.1.2 "Cooperate on standards, compete on implementation"

The AUTOSAR motto *"Cooperate on standards, compete on implementation"* explains which direction the world of automotive was taking perfectly. Competitors rather than playing off one against the other, they laid down the rules of the game in order to play a valid match.

So far, the entire work of AUTOSAR can be grouped in 5 phases: *Initialization*, *Phase I*, *Phase II*, *Phase III* and *Post Phase III* (table 1.2). Between 2002 and 2003, the *Initialization* stage involved: the discussion on objectives to follow, the development of the AUTOSAR platform and the definition of the guidelines to take up *Phase I*. Whereas between 2003 and 2006, the *Phase I* saw the realization of first tools, generators and 42 out of 46 Basic software components. During *Phase II*, the delivery of the release 3.0, 3.1 and 4.0 broadened the AUTOSAR architecture and methodology through the integration of new features such as functional safety and communication. Over the *Phase III* period, release 3.2, 4.1 and 4.2 focused on maintenance and selected improvements, in particular, partial networking and compatibility analysis. Finally, the release 4.3.0 (belonging to the *Post Phase III*) extended cryptographic functionality and made further improvements on Diagnostic information and Rapid Prototyping. Regarding the role of AUTOSAR in our days, goals and objectives will be discussed in section

1.3.

What is clearly evident from the table 1.2 is that the number of AUTOSAR members has undergone a remarkable upward trend throughout the whole period from the beginning to our days. Since the Phase I, the worldwide automotive standard has rocketed the components of its partnership. In fact, they went from 113 to 262 companies, within 15 years, and they are expected to keep rising in the foreseeable future.

Phase	Members
Initialization	6
Phase I	113
Phase II	166
Phase III	170
Post Phase III	191
Today	262

Table 1.2: AUTOSAR Phases and Members.

However, it is worthwhile to point out that companies of this worldwide partnership play different roles according to their position in the AUTOSAR hierarchy that is divided in 4 main groups (figure 1.1):

CORE PARTNERS They are 9 companies: BMW Group, Bosch, Continental, Daimler, Ford, General Motors, PSA Group, Toyota and Volkswagen Group. All of them have organizational, administrative and control responsibilities and their inner structure is organized in 5 working groups:

1. Executive Board: deals with the definition of the overall strategy.
2. Steering Committee: is responsible for the coordination of non-technical operations and planning of long term goals.
3. Project Leader Team: devises financial plans and rules for the working groups.
4. Legal Team: focuses on legal issues.
5. Communication Team: manages internal and external communications (press, web-site, etc.).

Each singular team meets regularly in order to define AUTOSAR specifications that are handled by Work Packages (*WPs*). Members of *WPs* are not only part of the Core Partners but also to other groups.

PREMIUM MEMBERS are part of the *WPs* and have access to current information.

ASSOCIATE MEMBERS have access to finalized documents.

DEVELOPMENT MEMBERS are part of the *WPs*.

ATTENDES even though they are part of the *WPs*, attendes participate through academic collaboration and non-commercial projects.

Figure 1.1: Autosar Members

1.2 The AUTOSAR Backbone

Although in the first part outlined the idea and the change behind AUTOSAR, we hardly mentioned the real core and bedrock of this partnership. Therefore, this section will give technical details to answer to questions such as 'How was a so abrupt change in the automotive industry possible without wasting time and money?' or 'How has the integration of external solutions to the AUTOSAR architecture been possible?' and, finally, 'How does the model take care of ECU and System constraints during the software development process?'.

All these questions find an answer through the descriptions of the four AUTOSAR milstones: architecure, methodology, application interface and acceptance tests.

1.2.1 Architecture

The AUTOSAR architecture consists of a software structure that can be split in two: the part below and above the RTE (figure 1.2). While the latter is mainly employed for the development and integration of software applications, the former focuses on providing low-level services. Obviously, the interaction among these parts identifies a central difference with preceding results. In fact, earlier applications were able to access to low-level services directly. Whereas, AUTOSAR introduces a middle layer (the RTE) that gives the green light to software components for the use of module interfaces.

It's important to stress the concept that AUTOSAR is not an operating system but a standard that defines the software architecture of an ECU. Moreover, what we explained can be compared to the two major architectures of an operating system:

Flat Architecture The OS components are functions that can be invoked by any application. The critical drawback is that a crush could spread over the whole system.

Layered Architecture The OS is divided into several layers and each of them

have a dedicated virtual address space. In comparison to the flat architecture, a malicious program cannot damage the kernel space.

The transition from a flat to a layered architecture has already happened in computer science history. At the beginning of 1980, a group of four programmers developed MS-DOS. This was an operating system based on flat architecture in order to leverage the least space and provide most functionality. Although it was designed just for few people, it became very popular in that period. However, its low level of kernel protection together with the chance of any application to access basic software services made MS-DOS vulnerable to errant programs. Consequently, to decrease system crashes coming from user applications, programmers decided that the following MS-DOS releases should be based on a layered architecture. To sum up, it is evident that AUTOSAR carries out a well-known and tested solution with the purpose of adapting it to a new scenario.

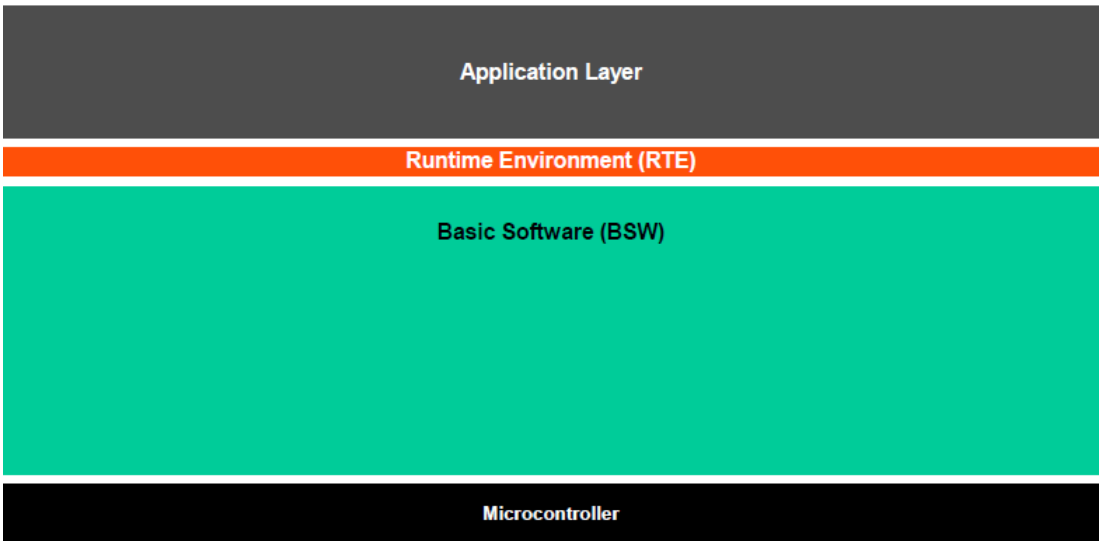


Figure 1.2: AUTOSAR Architecture overview

1.2.1.1 Basic-Software

The Basic-Software (BSW) is the layer above the micro-controller that provides services that depend on the hardware resources of the ECU. Overall, it is organized in modules that are gathered in a fixed number of layers. During the migration to AUTOSAR, one of the greatest issues to ascertain concerned how to fasten the migration without implementing all software modules and, consequently, wasting time. In order to overcome this obstacle, the automotive consortium focused on developing AUTOSAR-conform interfaces able to access to each single module without changing its content. However, what is an interface?

This term basically describes a program that has the fundamental role not only to abstract the hardware of a specific device but also to give access to the specific device regardless of its hardware realization and the number of existing devices of the same type []. An interface is made up with a set of data types, application programming interface (API) functions and error codes returned by the API.

In order to develop proper interfaces, AUTOSAR defines three implementation conformance classes (ICCs) to cluster BSW modules:

1. ICC1: It is the 'lowest' implementation class. The RTE and the whole BSW are put into one cluster. Only the interface between RTE and the ASW and the interface to the bus must be AUTOSAR-conform.

2. ICC2: Modules and RTE are gathered into separate clusters. ICC2 enables the integration of module coming from different vendors. For instance, the communication stack from the vendor A and the operation system from vendor B.

3. ICC3: This is the 'highest' class. In fact, all BSW modules have their own interface.

Moreover, it is worth underlining that modules communicate with other modules in compliance to well-defined rules. In particular, they can call interfaces horizontally or vertically. What is noticeable from the table 1.3 is that any layer could access to the interfaces of the SW layer below but bypassing one or more layers vertically is forbidden. In addition, horizontal interfaces are allowed for the Service and ECU abstraction layer. On the contrary, horizontal interfaces are not allowed for the micro-controller layer. Whereas, Complex Driver may get access to all BSW modules.

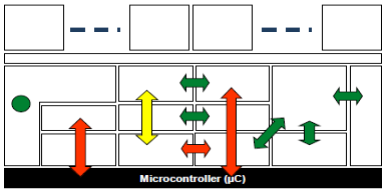
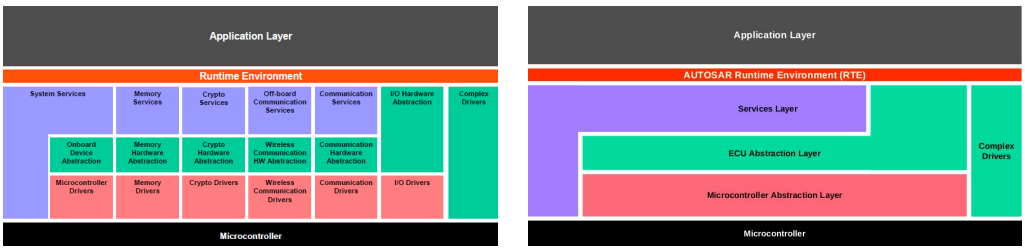


Table 1.3: BSW interface rules.



(a) *View of the BSW modules.* **(b)** *View of the four layers*

Figure 1.3: BSW structure.

Concerning the BSW layout, it consists of four layers (figure 1.3b):

Micro-Controller Abstraction Layer is the lowest software of the BSW that is responsible for making higher layers independent from the micro-controller. It mainly contains internal drivers [3] and it is divided in four modules: micro-controller drivers, memory drivers, communication drivers and I/O drivers.

ECU Abstraction Layer is above the micro-controller abstraction layer and offers an API to access to peripherals and devices. In comparison to the the previous layer, it includes external drivers [4] and the structure is composed

[3] Internal Drivers: drivers located in the micro-controller.
[4] External Drivers: driver that interact with external devices.

by four modules: on-board device abstraction, memory hardware abstraction, communication hardware abstraction and I/O hardware abstraction.

Service Layer is between the ECU abstraction layer, the RTE and the microcontroller. Even though this is the 'highest' layer of the BSW, part of the service layer communicates with the mictro-controller in order to allow the AUTOSAR OS and the BSW scheduler to work with the hardware. Operating system functionality, vehicle network communication, ECU static management, memory and diagnostic services are provided by this layer.

Complex Device Driver covers the right hand-side of the figure 1.3(b) and allows integration of device drivers that are not specified in AUTOSAR.

All in all, the BSW aims to provide basic services as:

- Input/Output: standardized access to sensors, actuators and ECU on board pheripherals.
- Memory: standardized access to internal and external memory.
- Crypto: standardized access to cryptographic primitives.
- Communication: standardized access to vehicle network systems, ECU on-board communication systems and ECU internal SW.
- Off-board Communication: standardized access to Vehicle-to-X communication, in vehicle wireless network systems and ECU off-board communication systems.
- System: ECU services and library functions.

1.2.1.2 Application Layer

The first noticeable difference between BSW and Application Layer is the architecture. While the former is based on a layered structure, the latter shows a component architecture. Overall, a component represents partially or completely a vehicle functionality that can interact with other components through ports.

However, the real benefit of ASW is the chance to implement multiple instances of the same component in a vehicle system. According to the figure 1.4, the instance of the *SeatHeatingControl* component is called twice for both front right and left seat. Even though they are in two different memory locations, instances share the same piece of code.

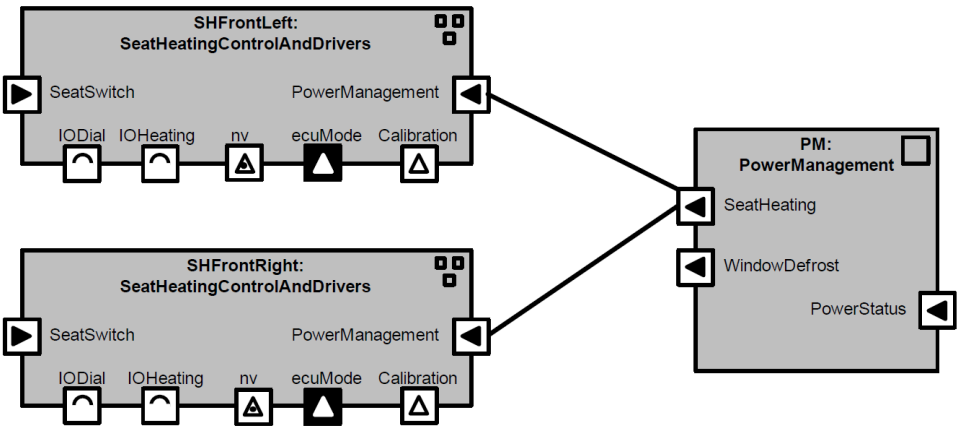


Figure 1.4: ASW SeatHeatingControl

Ports and Connectors

A port is the interaction point between two or more components. Each port is matched with a port-interface that describes data or service required or provided by an ASW []. There are five types of port-interface:

Client-Server Both client and server are software components. The Server provides operation that are invoked by several clients. For instance, the clients A and B may ask for the same function to the server that will return the output on the communication system.

Sender-Receiver this port-interface deals with the data exchange between two or more components. In comparison to the previous interface where the server offered an operation, now a sender forwards information to receivers. The data-elements goes from integers to complex data types such as strings and arrays. Any type of data exchange is logically atomic [5]

Non Volatile Data Interface it gives access to a non volatile (NV) element.

Calibration the entry point for modules to calibration parameters.

On the whole, there are two kinds of port: RPort and PPort. Intuitively, R and P indicate a port that receives and provides an element[6], respectively. However, if the origin or destination of an element is an AUTOSAR service, the port will be black otherwise the background colour is white.

RPorts and PPorts connection occurs through assembly-connectors that carries the request or the output to the right direction. Especially, the dependency of a component on other components leads to the difference between atomic component and composition. While the former does not rely on other components, a composition consists of a set of components (also defined as *prototypes*) and connectors that are brought together in a single ASW. Obviously, a composition

[5] Atomic Operation: It is an operation that is performed entirely or is not performed at all. In our case, this means that data mustn't be sent partially.

[6] Element is either a data or an operation.

could be the prototype of a larger composition.

In order to clarify the inner structure of a composition, the figure 1.5 shows an average composition. In this case, the *SeatHeatingControlandDrivers* has 7 Rports and it is made up with 3 prototypes (*ShDial*, *SeatHeatingControl* and *SeatHeating*) which exchange data such as LED value, position and HeatingELement.

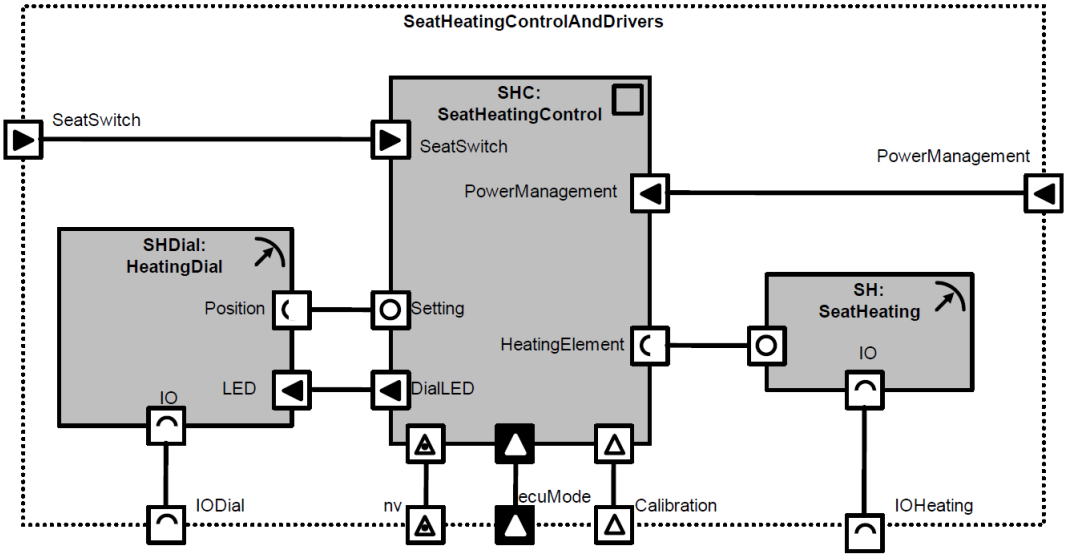


Figure 1.5: AUTOSAR composition

Components

Software components are distinguished in several types according to their functionality:

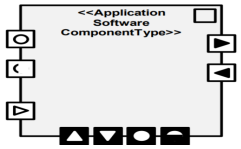
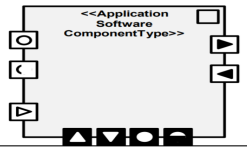
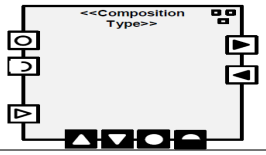
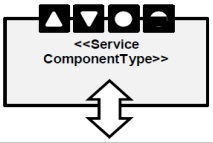
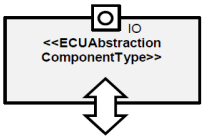
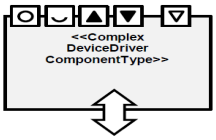
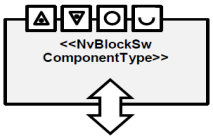
Application Software Component	The ASW is an atomic software component that describes a vehicle application. It interacts with **Sensor/Actuator Software Component** to use sensor and actuator data.	
Sensor/Actuator Software Component	Similarly to an ASW, this is an atomic software component that handles data and of operation coming from sensors or actuators.	
Calibration Parameter Component	It forwards data related to a calibration parameter.	
Composition	It describes partially or completely a vehicle function.	
Service Component	Component that is directly connected to a BSW module and provides only standardized services.	
ECU-Abstraction Component	It gives access to the ECU IO functionality.	
Complex Device Driver Component	This is a generalization of the ECU-Abstraction Component.	
NVBlock Software Component	This component furnishes an entry point for non volatile data at virtual functional bus level that will be explained in the following.	

Table 1.4: AUTOSAR components

1.2.1.3 AUTOSAR Runtime Environment and Virtual Functional Bus

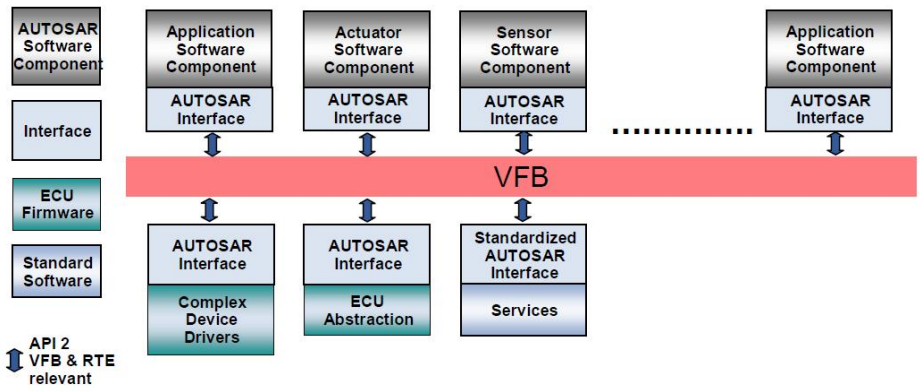


Figure 1.6: VFB connection.

The real communication between components on the bus is not achieved immediately. In order to establish a well-defined and robust path, there must be specific steps to follow. Therefore, AUTOSAR defines two levels for testing the software component connection.

The virtual functional bus (VFB) describes a "system modelling and communication concept" that creates a high level infrastructure for components. The VFB purposes are either simulating the real BUS functionality and verifying the connection among components in a virtual environment. Due to the fact that it does not rely on the hardware, the VFB includes some virtual low-level services such as Complex Device Drivers, ECU Abstraction and Services (figure 1.6).

On the other hand, the AUTOSAR runtime environment (RTE) can be defined as a real implementation of the VFB idea on a single ECU but with the aim of providing a complete environment. The adjective 'complete' indicates the presence of mechanisms that are not taken into consideration by the VFB. In fact, the RTE can invoke components according to scheduling algorithms, trigger events and provide mechanisms for the synchronization of components to shared resources.

The most noticeable difference between RTE and VFB is the number of provided services. It is undeniable that throughout the transition from a virtual to a real situation, components may require functionality of all BSW-services. For this reason, the RTE allows ASWs to interact with any BSW interface and to exploit additional resources such as CPU and memory. Moreover, a further contrast regards the way whereby the interaction between components happens. As you can see from the figure 1.7, the VFB mapping is realized through ports and connectors whereas the RTE employs Intra-ECU [7] and Inter-ECU [8] mechanisms.

[7]Intra: the communication occurs among components of the same ECU. Therefore, there are not external links.

[8]Inter: the communication occurs between components of tho different ECU. There is a link on the bus.

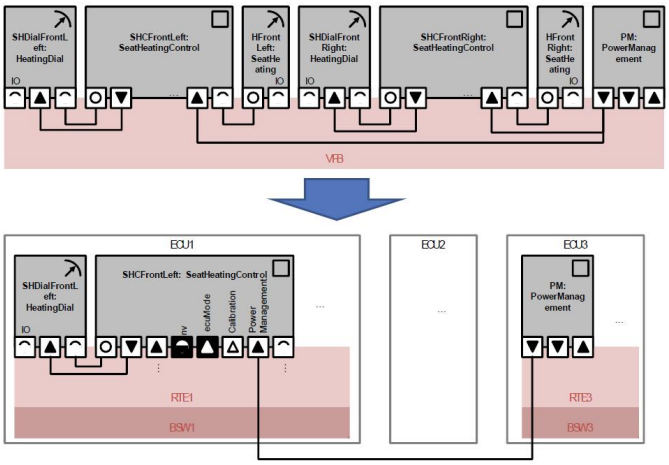


Figure 1.7: Transition from VFB to RTE.

Runnable Entity

Similarly to the link between PC program and process, the RTE does not invoke the entire ASW but only part of it. The piece of code that partially describes an ASW is called runnable entity or simply runnable. More precisely, it is a is a sequence of instructions that can be executed by the micro-controller, scheduled by the operating system and invoked by the RTE. Consequently, whatever is the type of ASW, there will be several runnables entities to describe its behaviour. Overall, there are two types of runnables:

1. Type 1: set of instructions that return in a finite time.
2. Type 2: set of instructions that finish when an external event is triggered.

Even though the concept of runnable is present in the RTE, the VFB and OS environment, their views are slightly different. From the RTE perspective, runnables that can run in the same context are clustered in the same task while the VFB sees a runnable as a sequence of instruction ready to be executed. On the other hand, the AUTOSAR operting system schedules the task according to specific algorithm (figure 1.8).

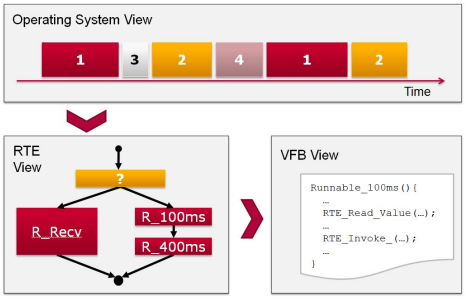


Figure 1.8: OS, RTE and VFB perspectives of a runnable

Aspect	**VFB**	**RTE**
Application View	API	API
System View	Centralized	Distributed
Communication	Ports	Intra and Inter ECU
Scheduling	Runnable	OS Task

Table 1.5: Difference between VFB and RTE.

1.2.2 Methodology

The second crucial pillar of the AUTOSAR standard is the methodology. It describes the dependencies of activities in a work-product flow regardless of when and how information are available. The Methodology simply outlines the availability of product in a work-product chain without defining the steps to follow or the timeline to respect.

While the methodology gives a wide overview of the whole process, the meta-model focuses on *how* something is defined. In AUTOSAR, a meta-model is a standardized XML file that is employed not only for technical information about electronic systems but also for the description of the software development process.

Moreover, it is important to bear in mind that the AUTOSAR architecture and methodology are not separate parties, on the contrary, they are strongly interconnected. In fact, the methodology requires as initial input a deep description of the system that is provided by the architecture.

Nomencalature

Before describing each singular stage of the Methodology, it is undeniable that the meaning of the symbols needs to be analyzed in advance for a clearer comprehension of the whole.

Icon	Name	Description
	Work-Product	This is a piece of information that could be the output or the input of an activity. Usually, in AUTOSAR it has a XML extension.
	Activity	It contains tasks and actions associated to a work-product.
	Guidance	It includes further instructions to an activity.
	Flow of work-product	This is not properly an entity but it defines the link between work-products and activities. According to the figure, the arrow illustrates the direction from the input to the output.

Table 1.6: Types of methodology components

AUTOSAR Methodology Chain

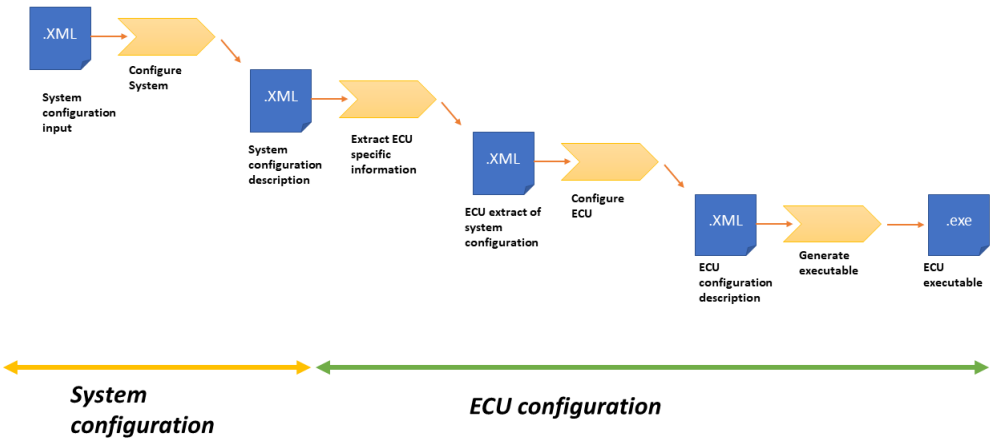


Figure 1.9: AUTOSAR methodology

The whole chain can be split in two essential subsets: the system and the ECU configuration. While the former includes the left-hand side of the flow that goes from the *System Configuration Input* to the *System Configuration Description*, the latter describes the ECU design step whose final output is an executable file

ready to be loaded on the exact ECU.

The *System Configuration Input* is a work-product which provides information about: software components, ECU resources and system constraints [9]. Whereas, the *Configure System* activity illustrates the exact mapping of ASW components to the ECUs according to resources and timing requirements. The output of this activity is the *System Configuration Description* that contains all system information and defines the entry point for the ECU design part.

In that part, the *Extract ECU Specific Information* activity picks out just all the information associated to a specific ECU from the *System Configuration Description* and stores them inside the *ECU Extract of System Configuration* file. Afterwards, the *Configure ECU* caters to all BSW modules requirements and returns the *ECU Configuration Description* file. Finally, this file is utilized as an input of the *Generate Executable* activity for the generation of the *ECU Executable* that is a piece of software describing the BSW, the RTE and the connection among components.

1.2.3 Application Interface

In order to establish a correct communication among SWC that belong to different automotive domains, the interfaces should be able to process response and requests via API correctly.

This process is guaranteed by the Application Interface (AI) table whose main goal is to publish a well-defined and robust list of AIs. In fact, the AI table is an excel file which specifies interfaces of several domains in terms of syntax and semantics. Regarding the structure, it is organized in several working sheets containing information about:

- Compositions from each domain
- Components
- Ports
- PortInterfaces
- Data Types
- Units
- Instances of component types

[9] System constraints: they mainly refer to limits of the bus signals, topology and mapping. These constraints forbid to map a software component to the incorrect ECU.

1.2.4 Conformance Test

As mentioned at the beginning of the chapter, one of the most important AUTOSAR purposes is to ease the collaboration between manufacturers and suppliers. One of the advantages of a layered architecture is the chance for the automaker to buy products from different suppliers and, then, to integrate the various solutions in the same system. However, it is undeniable that automakers need to verify if the purchased software is AUTOSAR-conform. This issue is addressed by the *Conformance Test*. Rather than attesting the functional correctness of a product, it aims to verify its conformance through tests that relies on the kind of ICC.

Although conformance and integration test seem to be similar procedures, there are striking differences. In particular, the importance of the former is to assure that the integration phase occurs among AUTOSAR-conform piece of software. Thereby, a more efficient and reusable solution for the automaker is guaranteed. Whereas, the integration test makes all the software solutions interact properly. However, at the end of the conformance process (diagram below), the results of the conformance tests are verified by an external party which is the *Conformance Test Agency* (CTA):

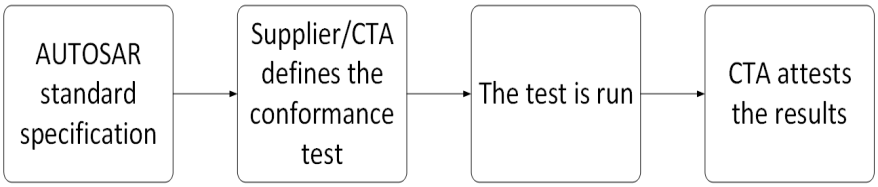


Figure 1.10: Conformance test flow.

1.3 The Future of AUTOSAR

The development of the AUTOSAR standard has entailed enormous benefits in the world of automotive and dramatic change from a business point of view. In fact, the transition to a layered software architecture has enabled automakers to widen the range of choice for each part of the system.

However, in the forseeable future the AUTOSAR architecture will undergo eveident changes. In fact, the study conducted in this chapter regards the AUTOSAR *Classic Platform* (*CP*) but AUTOSAR is expected to move towards a new type of architecture that is the *Adaptive Platform* (*AP*). There are several reasons behind this migration. First, the benefits related to short development cycles, inter-operability of the applications and a higher quality. Secondly, the need to meet demanding requirements of vehicle functions in terms of computing resources and safety. It is worthwhile to highlight that the *AP* will not take the place of either the *CP* or non-AUTOSAR platform but it will communicate with them (figure 1.11).

A further consideration regards future applications, in comparison to actual ASWs that are based on electro-mechanical system that don't go through evident modification during the automobile life-span, future applications will rely on complex algorithms that will need to be kept up-to-date frequently. Therefore, software change is prone to occur often during the life-cycle of a vehicle. For instance, the autonomous driving, which is one of the greatest challenges of the automotive future, depends upon functionality such as 'image detection' or 'path planning' whose algorithms are likely to be updated periodically.

Obviously, the platform upgrade needs to be supported by the hardware of the ECU. Although ECUs are made up with multicore processors, their features have never been exploited by the *CP* at all. Consequently, one of the aims of the AP is to leverage the ECU potential through decisive key-factors like parallel processing, safety and security.

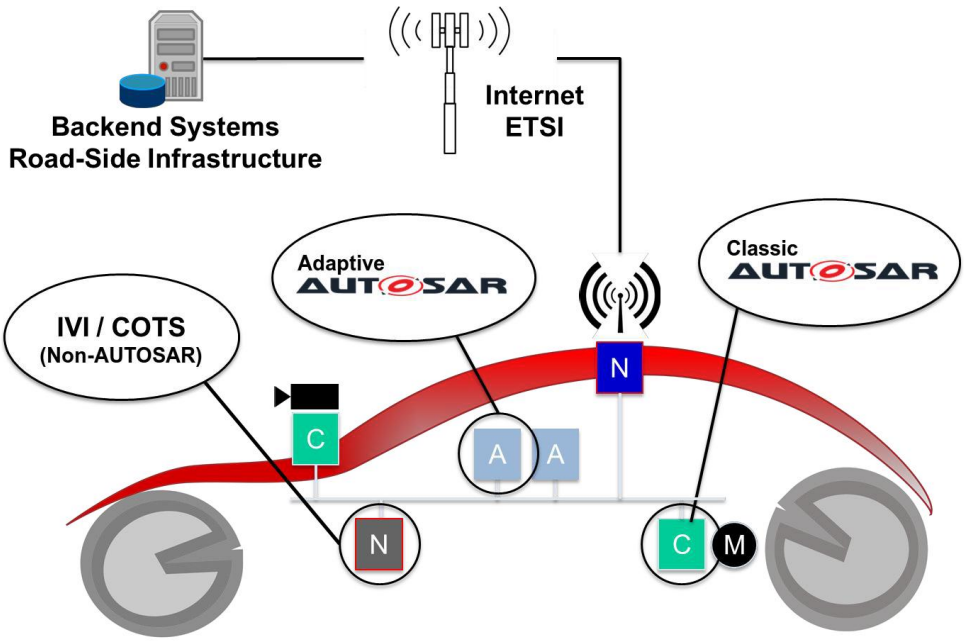


Figure 1.11: Platforms Integration

Chapter 2

Fuel Level Control

Fuel Level Control

The most remarkable strength of a vehicle has always been its ability to adapt and integrate new technologies to previous versions. This has enabled carmakers to increase the order of complexity gradually and to make the automobile the advanced means of transport of our days. Since the first production of a wheeled conveyance as a gift for the Chinese emperor in 1762, the automotive world has undergone evident change especially from a business view. In fact, after Henry Ford made the *Ford Motor* become one of the most profitable companies, the automotive industry has developed into one of the bedrocks for the economies of wealthier countries.

However, it is interesting to note that the center of attention in car production has changed throughout the decades. In fact, it went from the realization of powerful engines, over the whole 19th and 20th century, to the integration of software and hardware components in the last 30 years. This migration has led C code functions to become more and more crucial in a car development up to take the place of mechanical applications. For instance, the comparisons of the three different implementations of the same *throttle command* in the figure below, outlines how electronics has affected vehicle function. In comparison to the past when a long arm made the butterfly valve rotate, today a sensor detects the slope of the gas pedal and transmits the data to the *Engine Control Unit* which controls the butterfly valve rotation.

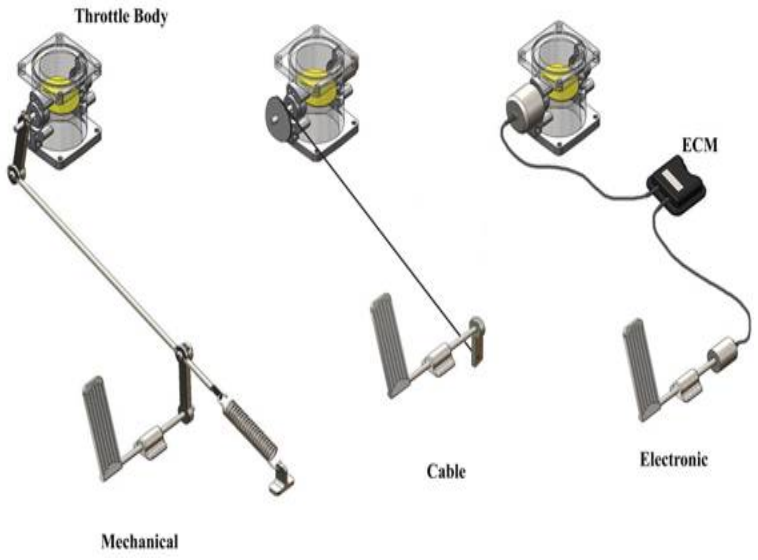


Figure 2.1: Evolution of the *throttle command*.

2.1 The new perspective of vehicle function

The deep view of AUTOSAR standard and, in particular, the description of the three layers that comprise its architecture identify a connection point between the previous and the present chapter. In fact, a vehicle function (VF) can be described not only as an automotive service that is provided by the system to the driver but also as an application developed in the Application Layer through ASWs. Overall, a VF is classified according to the level of complexity and the priority order of the information that forwards on the vehicle network. While the former refers to stand-alone or a complex function whose inputs could come from other VFs, the latter lays down the execution order among VFs in the interest of avoiding harmful consequences.

Regarding complex VF, today the *Electronic Stability Program* (ESP) has the crucial role in computing the vehicle speed and sends this data to other VFs such as the *Air-Bag*, the *Autonomous Car Control* (ACC) or, simply, the GPS. Whereas, concerning the importance of priorities, what is inferable from the table 1.1 is that functions of ECUs of the *PowerTrain* and *Vehicle Safety* sub-domains are likely to take precedence over VFs belonging to other domains.

Moreover, a real time[1] environment has to guarantee adequate response time and forbid the priority inversion problem[2]. In order to cope these issues, data transfer rate and well-defined protocols [3] need to be properly devised and implemented. In fact, it is undeniable that they have a huge impact in an efficient communication. For this reason, the organization of the vehicle BUS in five classes allows the system to match the application content with the necessary transfer rate and, consequently, the right priority (table 2.1).

However, introducing five classes and four protocols means dealing with five different message structures. Bearing in mind the *ESP* example, one immediately wonders how to make all VFs receive the correct data if the receivers belong to distinct classes. Because data are not formatted in a common always-decipherable way, a gateway is employed on the BUS to accomplish that purpose (figure 2.2). This is a hardware device whose task is similar to an interpreter. In fact, once received the data from the BUS, the gateway converts it in another format and forwards the message to the ECUs belonging to other classes. In this way, the same variable is readable from all the VFs without mistakes.

[1]Real time system: it is a system whose tasks has to respond correctly within the deadline.
[2]Priority Inversion: some tasks are blocked by others whose priority is lower.
[3]Protocol: Set of rules for data exchange or transmission.

Protocol	Class	Transfer Rate	Description
LIN	A	10 kb\s	The **L**ocal **I**nterconnect **N**etwork is a local subsystem that guarantees the vehicle network in a demarcated area of the vehicle such as light, door and window control. This is a cheaper solution in comparison to the *CAN* and it is based on a master-slave approach.
CAN	B	125 kb\s	The **C**ontrol **A**rea **N**etwork was the first bus system in a vehicle. In this case, the *CAN-B* also defined as Low-Speed CAN is specifically employed for comfort and body applications like control of air-conditioning, seat adjustment or mirror adjuster.
	C	1 Mb\s	The *CAN-C* or High-Speed CAN aims to meet real-time constraints on Powertrain applications such as electronic transmission control and engine management system.
FlexRay	C+	10 Mb\s	Altough the FlexRay provides approximately the same services of the *CAN-C*, there are crucial differences in terms of response time, "composability" and flexibility (analyzed in depth [bosch reference]). However, experts say that due to FlexRay is still in development, the silver bullet may be a combination of these two protocol.
MOST	D	>10 Mb\s	The **M**edia **O**riented **S**ystems **T**ransport is a protocol whose data rates is so large because it is used only for infotainment and telematics applications.

Table 2.1: Protocols and Classes on the BUS

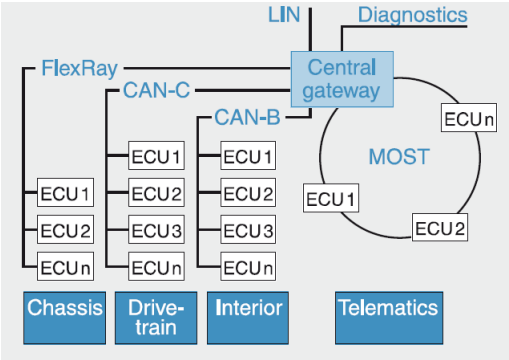


Figure 2.2: Bus communication

Within this context, the *'Fuel Level Control'* is a periodical vehicle function of the Infotainment area with the aim of providing a reliable percentage of the fuel regardless of any external dynamics. Although this task seems to be easier in comparison to other functions, there are several situations to keep uppermost in mind. Usually, the reason behind a wrong representation of the fuel percentage could be by either a sensor failure or an external scenario. While the former is coped asserting a variable which confirms whether the sensor measurement is acceptable, the latter needs a robust logic able to distinguish a real decrease from a temporary variation of the fuel. In that case, the algorithm has to go over three main issues:

1. **Fuel-Sloshing**: this phenomenon generates repeated waves inside fuel-tank. It could be the consequence of a street in bad condition or one with a steep slope.

2. **Sensor Positioning**: It deals with the position of the sensors in the fuel-tank. Obviously, this issue strictly depends upon the shape of fuel-tank. In case of an *Alfa Romeo Stelvio* which is horseshoe in shape, there must be two sensors inside.

3. **Not Sensibility Range**: It happens when the fuel level exceeds the upper bound or is inferior than the lower bound, 100% and 0% respectively. In this situation, sensor data are not completely trustworthy.

2.2 Sensor Handling

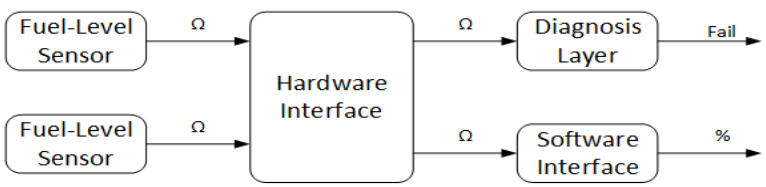


Figure 2.3: Path from fuel-sensors to the VF.

A modern vehicle includes thousands of sensors whose goals are to turn measures in electrical signals for ECUs. As indicated in the figure 2.3, before becoming a

variable for a vehicle function, signals need to follow a well-defined path that in case of the *Fuel Level Control* consists of four steps. The hardware (HW) and software (SW) interfaces address issues related to the 'cleaning' of the raw signal and the conversion of the resistance value to a percentage. Whereas, the Diagnosis Layer attests if the sensor data is out of range. Eventually, the percentage of the fuel level and a fail variable are forwarded to the *Measurement Logic* where the entire algorithm is implemented.

2.2.1 Fuel-Level Sensor

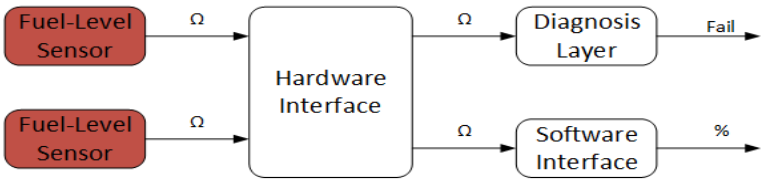
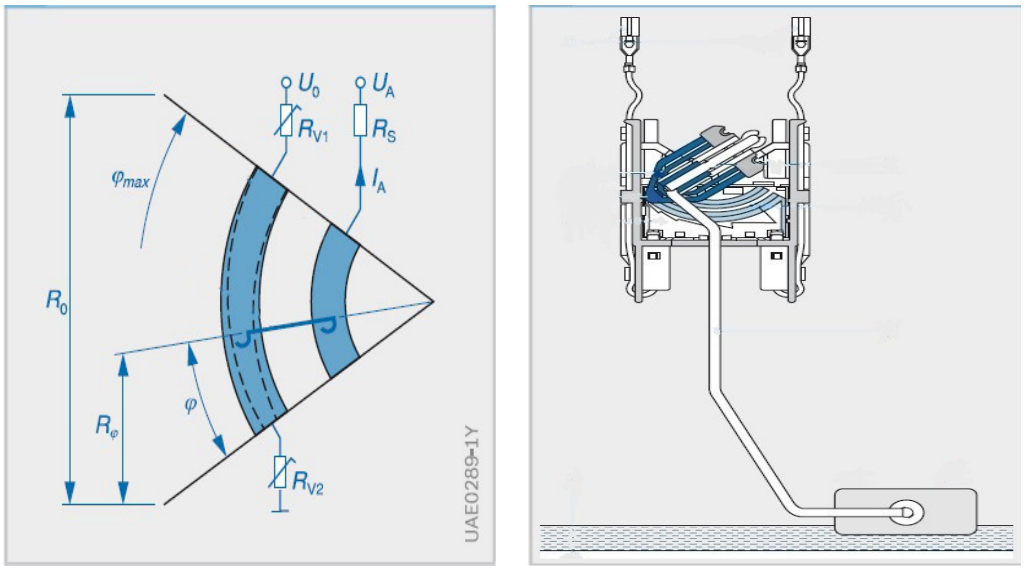


Figure 2.4: Fuel-Level sensors.

The reasons why position sensors are widely employed in a vehicle is not only for their property to record and detect any angular or linear displacement but also the longer life-span in comparison to other sensors. On the other hand, one of the disadvantages is the complication to be miniaturized due to the displacement is an extensive variable that that depends on the size of the system (table 3.2).

It is important to note that the high number of ways whereby the measurement is performed leads to several types of position sensors. In our case, *Fuel Level Sensor* belongs to the potentiometer type which is a kind of sensor able to detect any displacement through resistance values. The scientific principle that describes its behaviour is the *wiper potentiometer*. More precisely, as depicted in figure 2.5(a), the wiper extremities are always in contact with two circular resistance tracks. While the inner track is made up with approximately 100 resistive pads in a range between 50 and 1000 ohm and aims to provides the resistance value as a function of the angle, the outer circular area protects the sensor from overleading. Although the type and the quality of the fuel are crucial aspects that affect the wiper dynamics, a current $I_a \leq 1\,\text{mA}$ is applied to the circuit in order to keep wear as low as possible and guarantee contact cleaning.

However, the link between the wiper and fuel variation is achieved through an arm that connects the float to the wiper spring. In fact, the fuel-level sensor design consists of a 'sealed' and not-sealed part. The former is encapsulated in a waterproof and dust-proof tank that includes a potentiometer with wiper arm (wiper spring), printed conductors (twin-contact), resistor board (pcb) and electrical connection. Whereas, the float and the arm (wiper liver) are fully immersed in the fuel. When the fuel level changes, the arms moves horizontally and makes the wiper rotates along the resistance tracks.



(a) *Wiper potenziometer principle.* **(b)** *Fuel Level Sensor.*

Figure 2.5: Detailed view of the fuel-level sensor.

Advantages	***Disadvantages***
Cheap	Mechanical wear
Simple design	Problems with fluids
Temperature range	Expensive Testing
Calibration	No miniaturization
Assembly	Noise

Table 2.2: Advantages and disadvantages of potentiometer

2.2.2 Hardware Interface

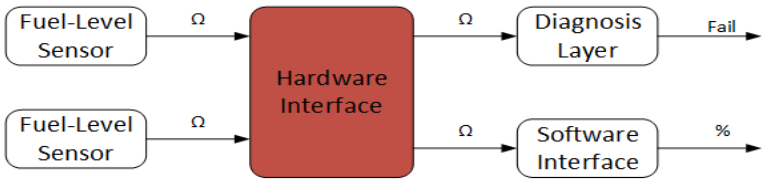


Figure 2.6: Hardware interface.

During the second phase, *HW interface* receives row values directly from the fuel level sensors and provides a clean data to both the *Diagnosis Layer* and the *SW interface*. It is undeniable that the frequent use of sensors, such as fuel-level sensors, require driving circuits able to tackle the contact aging issue. For this reason, the injected current from the electrical connection in figure 2.5(b) is driven by the pulse width modulation (PWM)[4] rather than leaving it to flow constantly inside the sensor circuit. The main advantages of this approach are

[4]PWM: it is a modulation process that regulates the power for electronic devices.

both to limit contact corrosion and to ensure a long-lasting reliable output.

2.2.3 Software Interface

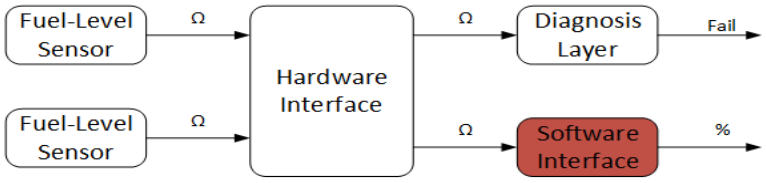


Figure 2.7: Software Interface.

In the final stage, data coming from the HW interface are managed via software approaches. In particular, the *SW interface* has the primary purpose to convert the resistance in a percentage. However, this conversion is not achieved immediately but it is figured out through a specific *calibration table* that takes into consideration a slight offset error. This introduces a thin difference between nominal and offset value. Thereby, the 100% and 0% are indicated for a while during a journey even though the fuel-tank is neither completely full nor empty. Obviously, the offset can be modified because it is a tuned parameter that depends on other vehicle features like model, weight or engine size.

2.2.4 Diagnosis Layer

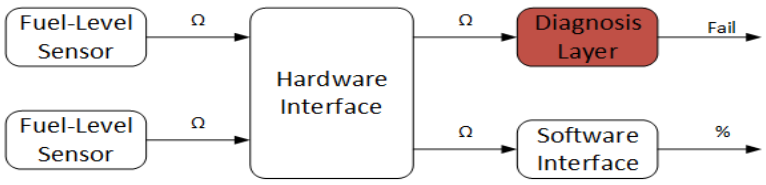


Figure 2.8: Diagnosis Layer.

Similarly to the SW interface, the *Diagnosis Layer* receives resistance value and detects whether the H interface output is affected by an error. If a short or open circuit is present for more than specific setting time, the error is validated and the percentage carried out by the *Software Interface* will be disregarded. However, the dynamics of the *Diagnosis Layer* will be reclaimed during the implementation of the FAULT condition of the vehicle function.

2.3 Vehicle Function Interconnections

According to the figure 1.11 at the end of the AUTOSAR chapter, an ECU commnicates on the BUS with other ECUs and sensors. Consequently, it is likely that

VF inputs include not only sensor data, as described before, but also variable related to other VFs.

In case of the *Fuel Level Control*, it is undeniable that the speed of the vehicle is a fundamental feature for the logic because it makes the fuel-level indicator decrease properly. In fact, the fuel consumption of a car that is accelerating at a constant speed of 30 km/h is different from another at 150 km/h. Although this aspect will be discussed in the next paragraph, it is worth outlining that the presence among the inputs of two variables, which indicate the 'present' fuel level through different computations, allows the logic to overcome issues like parking on slope, fuel loss or fuel sloshing. However, the purpose of this section is to provide a description of the all variables that are exchanged by the *Fuel Level Control*.

2.3.1 Inputs

Vehicle Speed : it shows the instantaneous speed of the vehicle in km/h. It is used to move from a static to a dynamic condition. Obviously, the speed affects how fast the fuel level decreases.

Vehicle Speed Fail Status : similarly to the output of the *Diagnosis Layer*, the *Vehicle Speed Fail Status* attests the worth of the vehicle speed.

Fuel Consumption : the *Fuel Consumption* comes from another vehicle function and provides the percentage of fuel level in %/h. Due to the fact that the fuel consumption and the output of the *Software Layer* contains the same information, the logic figures out a weighted average among these two inputs.

Fuel Consumption Fail Status : the *Fuel Consumption Fail Status* attests the worth of the fuel consumption.

Key Mode : The status of the key contains essential information for the execution of the vehicle function. However, for what concerns this context there are just three conditions to keep forefront in mind:

1. *Key ON*: The key is inserted and the vehicle control panel is turned on.
2. *Key OFF*: The key is not inserted and the whole vehicle is turned off.
3. *Key ON Engine ON*: The key is inserted and the engine is running.

Key Mode Fail status : the *Fuel Consumption Fail Status* attests the worth of the *Key Mode*.

Fuel Level Sensors : sensors provide the percentage of remaining fuel.

Fuel Level Sensors Fail Status : is one of the outputs of the *Sensor Management* that attests the worth of data coming from the sensors.

Reserve : is a threshold that is used to switch on the *Low Fuel Lamp*.

Soglia : when the absolute difference between two samples is higher than the *Soglia*, fuel loss is detected. Overall, this is not a fixed data but it can be changed by the supplier.

MLM min : minimum value to pass in the *MLM* strategy.

MLM max : minimum value to exit from the *MLM* strategy.

2.3.2 Outputs

Fuel Level : shows the computed fuel level percentage.

Low Fuel Lamp : if the percentage of fuel is lower than the *Reserve* threshold, the spy turns on so the driver detects the need to refuel.

Fuel Level Fail status : gives information about the reliability of the fuel level signal.

2.4 Measurement Logic

The *Measurement Logic* together with the *Sensor Management* comprise the body of the *Fuel Level Control*. While the latter handles sensor data, the former has the crucial task to deal with the implementation of the logic. It consists of four working conditions: STATIC, DYNAMIC, RIFO and FAULT.

Therefore, the vehicle function purpose, the fuel level sensor principles and the input/output description have been the necessary requirements for a detailed explanation of the algorithm. However, some details about the implementation will be hidden for company reasons.

2.4.1 STATIC

The STATIC measurement is the part of the logic that includes all the situations with the engine off. Essentially, it can be divided in ON and OFF status

whose transition is allowed by the presence of the key in the vehicle control panel. Clearly, this working condition takes into consideration only circumstances before or after a journey or mission.

Thinking about a real situation, when the driver inserts the key for the first time without rotating it (LOCK in figure 2.9), from a logic point of view, this condition is identical to a key removed from the panel. Therefore, the logic always starts from the OFF status of the STATIC condition. Here, all the local variables are initialized to a default value and results of previous missions are disregarded. During the second phase, the driver is likely to rotate the key to turn on the vehicle without starting the engine (ACC in figure 2.9). This provokes the transition from the OFF to the ON status and, consequently, the computation of the fuel level in a static situation for the first time.

Figure 2.9: Key status illustration.

The way whereby the computation is performed in the STATIC essentially is based on a comparison between two levels in two distinct time instants. As soon as the the ON state of the STATIC is active, a minimum sampling interval, during which the logic cannot move elsewhere, must be guaranteed. At the end of that interval, the logic keeps on sampling until the fuel percentage is sent to the control panel as a result of the mathematical average of the sensor data. Alternatively, there could be a transition in the OFF state. In this case, the sampling is not interrupted but continues in the OFF state as long as the end of the interval is reached.

In order to detect any fuel loss, the algorithm of the VF makes a comparison between the last computed value before the end of the journey and new computation at the beginning of the next drive. If the difference between two fuel level measurements are lower than *Soglia*, the percentage is updated and a scenario like parking on a slope is filtered out.

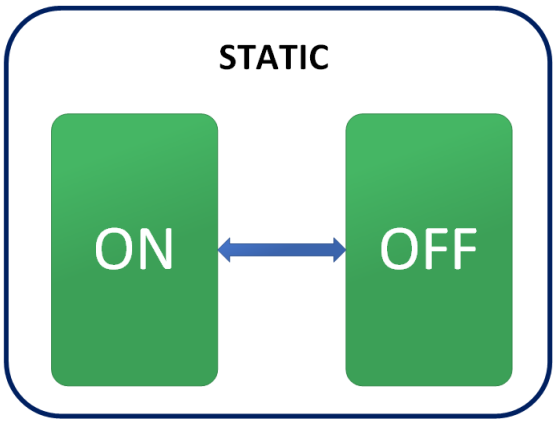


Figure 2.10: High-level representation of the STATIC state.

2.4.2 DYNAMIC

As soon as the driver turns on the automobile and, obviously, the engine is on (ON in the figure 2.9), the logic of the vehicle function goes from the STATIC to the DYNAMIC working condition. This is an essential part of the whole algorithm that deals with updating the fuel level while the car is running. The DYNAMIC comprises three filters: *Pre Filter*, *Speed Filter* and *MLM Filter*.

Pre Filter

Because the outputs of the *Sensor Measurements* are affected by interference, noise and contact chatter, a filter able to make the signal more reliable needs to be taken into account which is exactly the aim of the *Pre Filter*. This is a second order digital filter that receives as inputs the weighted average (2.1) and returns stable fuel level (figure 2.11). However, due to its quick dynamics and the short sampling interval of the ON state, it is inactivated in the STATIC. Regarding the weighted average, it is figured out through the formula:

$$Fuel_{Average} = Weight_1 * Sensor_{Level} + Weight_2 * FuelConsumption_{Level} \quad (2.1)$$

What is noticeable from the expression above is that it is the sum of two multiplications of a percentage coming from different sources and a mathematical weight. The weights have a huge impact in making a decision between the sensor and the *Fuel Consumption* input. Obviously, the weight choice relies on the quantity of fuel level inside the fuel tank. While the *Not Sensibility Range* (*NSR*) covers the whole percentage that goes from the "*high threshold*+10" to the "*low threshold*-10", the others are included in the *Sensibility Range* (*SR*). Although the idea of increasing the *high threshold* of 10% or lowering the *low threshold* of 10% appears illogical, it is worthwhile to stress the concepts that they are not nominal value and *high threshold* and *low threshold* can be chosen by the supplier.

Beyond the goal of providing a reliable input for the *Pre Filter*, the weighted average aims also to minimize the set of percentage belonging to the *SR*. In order

to reach this goal, the weights are defined differently within the *NSR*. In fact, if the fuel level exceeds the *high threshold* or lowers the *low threshold*, the interpolation method is performed. Whereas, when the fuel is out of the *NSR* the *Fuel Consumption* is disregarded and the weighted average is equal to the sensor data.

Range	Level	Weights
Not Sensibility Range	$HT{+}10 \geq FL \geq HT$	$Weight_1 = f(inter, WA)$ $Weight_2 = 1 - Weight_1$
	$HT \geq FL \geq LT$	$Weight_1 = 0$ $Weight_2 = 1$
	$LT - 10 \geq FL \geq LT$	$Weight_1 = f(inter, WA)$ $Weight_2 = 1 - Weight_1$
Sensibility Range	$(FL \geq HT + 10) \| (LT \leq LT - 10)$	$Weight_1 = 1$ $Weight_2 = 0$

Table 2.3: Fuel Level Range.

Figure 2.11: Pre-Filter scheme.

Speed-Filter

The *Speed Filter* is a first order transfer function that receives as an input the percentage calculated by the *Pre Filter* and estimates the fuel percentage according to the vehicle dynamics. This means that the filter behavior is not one-and-only but depends upon the vehicle speed. There are three possible actions:

- ***High Filtering***: The vehicle speed is higher than the speed threshold. This means that fuel will decrease quicker than the other circumstances.
- ***Low Filtering***: occurs either when the vehicle is below the speed threshold or the vehicle speed data is absent. In this scenario, $\tau_{low} \geq \tau_{high}$ because the fuel indicator decrease slower than the *High Filtering* situation.
- ***Refuel Filtering***: Regardless of the engine activation, in this condition the level of fuel rises and car is not moving. The time constant is set to $\tau_{refuel} \geq \tau_{low} \geq \tau_{high}$.

However, a further scenario to keep in mind concerns the failure of the vehicle speed input, the filter acts like a vehicle running at low speed.

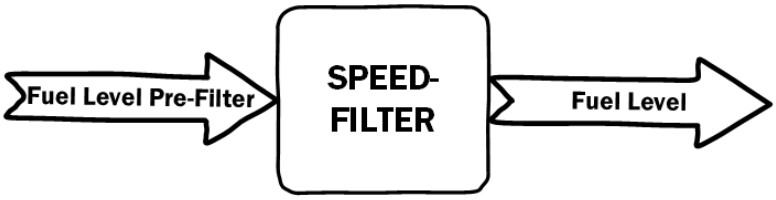


Figure 2.12: Speed-Filter scheme.

MLM Filter

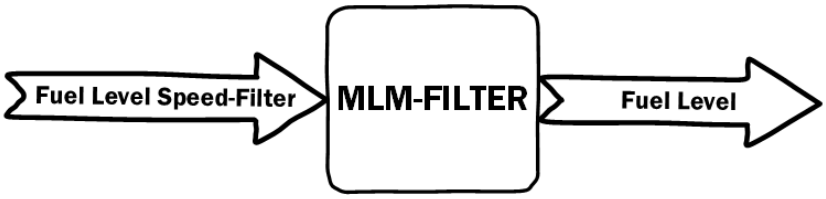


Figure 2.13: MLM-Filter scheme.

If the fuel level is lower than the *MLMmin*, the *Minimum Level Management* (*MLM*) is activated and the output is no longer computed by the *Speed Filter*. When the fuel-tank level is almost empty, the goal of this part is to force the fuel level indicator to reach zero within a certain time. However, in case of fuel loss, the 0% is reached more rapidly.

How these filters interact is described in the following figure:

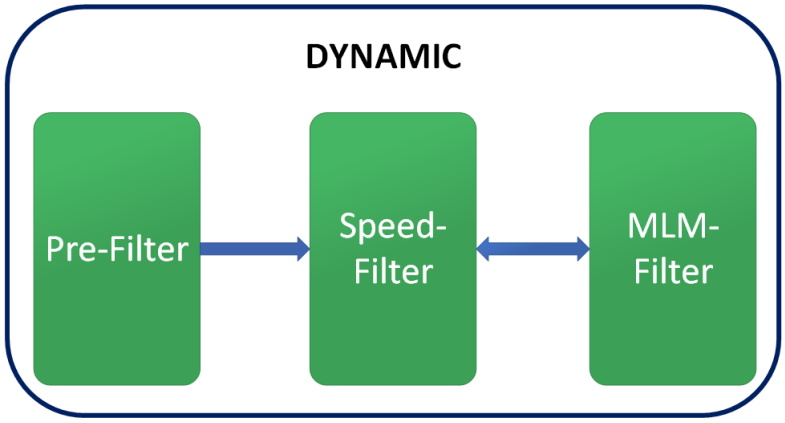


Figure 2.14: High-level representation of the DYNAMIC state.

2.4.3 RIFO

It is no doubt true that refuel covers a fundamental aspect throughout the journey. From the *Fuel Level Control* view, this means a new state where the fuel level tends to rise up to an unspecified percentage. The RIFO condition can be reached from both the STATIC and DYNAMIC. In fact, if a driver is waiting for the end of the refuel and switches on the vehicle control panel (*Key ON* case), from a logic point of view, there will be a direct transition from the STATIC to

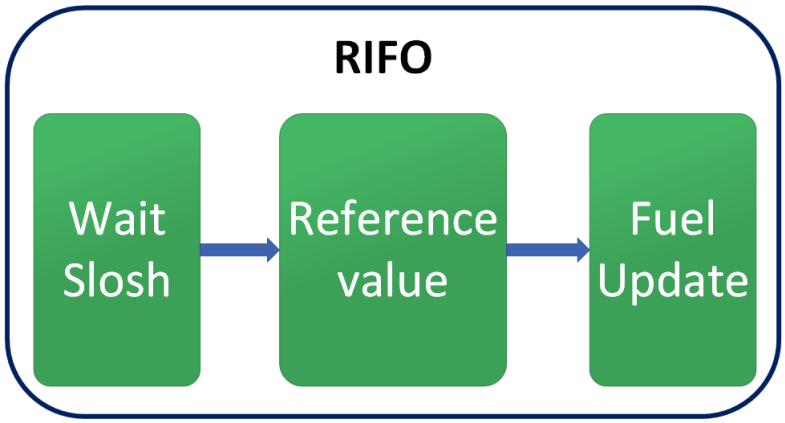


Figure 2.15: High-level representation of the DYNAMIC state.

the RIFO condition. Whereas, if the speed of the car becomes lower than a fixed threshold the transition comes from the DYNAMIC.

Because the car could reach 0 km\h decreasing the vehicle speed slower and slower or via a sudden hard braking, the first part of the RIFO is a dead time for diminishing possible fuel sloshing effect. Afterwards, the first ten samples of the sensors are used to compute the fuel level. This measurement can be defined as a landmark not only because it will not change as long as the RIFO is active but also it will be constantly compared to a second sampling that is performed every couple of seconds. Thereby, the *Fuel Level Control* output is updated regularly. However, entering in the RIFO working condition is not always associated with the refuel action. For instance, if a driver is stuck in the traffic jam, the car is on but the speed is zero most of the time. Hence, there will be continuous transitions from the DYNAMIC to the RIFO and vice-versa without any fuel level variation.

2.4.4 OUTPUT

Although the OUTPUT is not properly a true working condition, its functionality is incredibly crucial for the VF purpose. As mentioned earlier, STATIC, OUTPUT and RIFO description underlined the necessity of forwarding reliable information about the fuel percentage and, consequently, the fuel lamp status. In fact, if the fuel level exceeds the *Reserve* threshold, the *Low Fuel Lamp* is turned off after a while. Whereas, when the fuel level is lower than the *Reserve* threshold, the *Low Fuel Lamp* lights after a predefined amount of time.

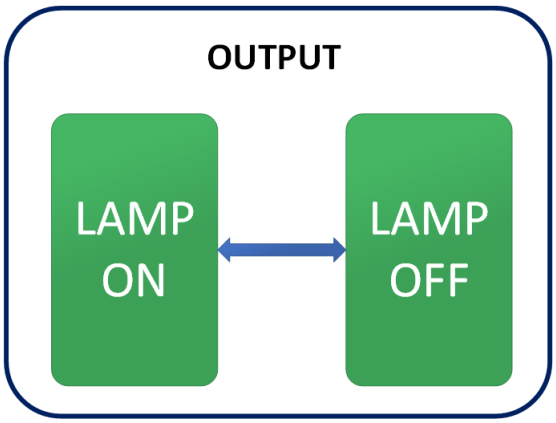


Figure 2.16: High-level representation of the DYNAMIC state.

2.4.5 FAULT

Last but not least, the FAULT condition deals with any error related to the input variable. Most input errors do not always represent a migration to the FAULT state simply because they are overcome by internal choice of the logic. For instance, if the *Key Mode Fail status* verifies a mistake of *Key Mode*, the algorithm continues its execution maintaining its last valid condition. Similarly, when *Vehicle Speed Fail Status* attests the error, the *Speed Filter* works in the low filtering mode and, finally, a wrong representation of the *Fuel Level* makes all measurements depend just on the fuel level sensors.

As a result, the migration to the FAULT state is provoked only by the assertion of the *Fuel Level Sensors Fail Status* variable. If this input remains true for the whole validation time, the sensor error is confirmed and the *Fuel Level* is set to zero. In case of *Key Off* event when the error is validated, the error is not canceled accidentally because the logic will move to an independent state of the OFF STATIC condition. In this way, at the following *Key On* event, the logic goes directly to the FAULT. On the other hand, the devalidation occurs if the *Fuel Level Sensors Fail Status* is false for a definite time interval.

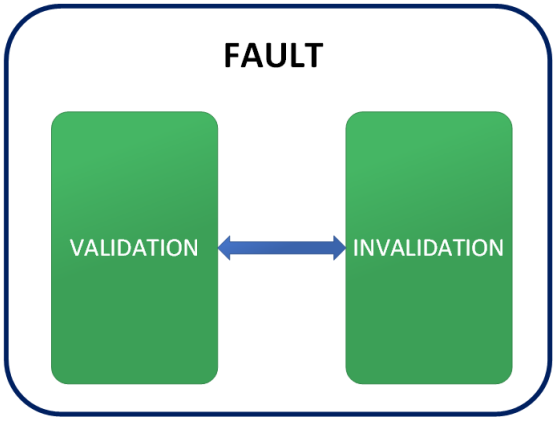


Figure 2.17: High-level representation of the FAULT state.

2.5 Vehicle Function Design

Similarly to thin pieces of a puzzle that need to be put together properly in order to produce a correct image. This section brings together each single working condition previously explained in order to provide an overview of the vehicle function and how the parts interact each other. Starting from this preliminary sketch, the next chapter will deal with the development of the vehicle function via the model based approach.

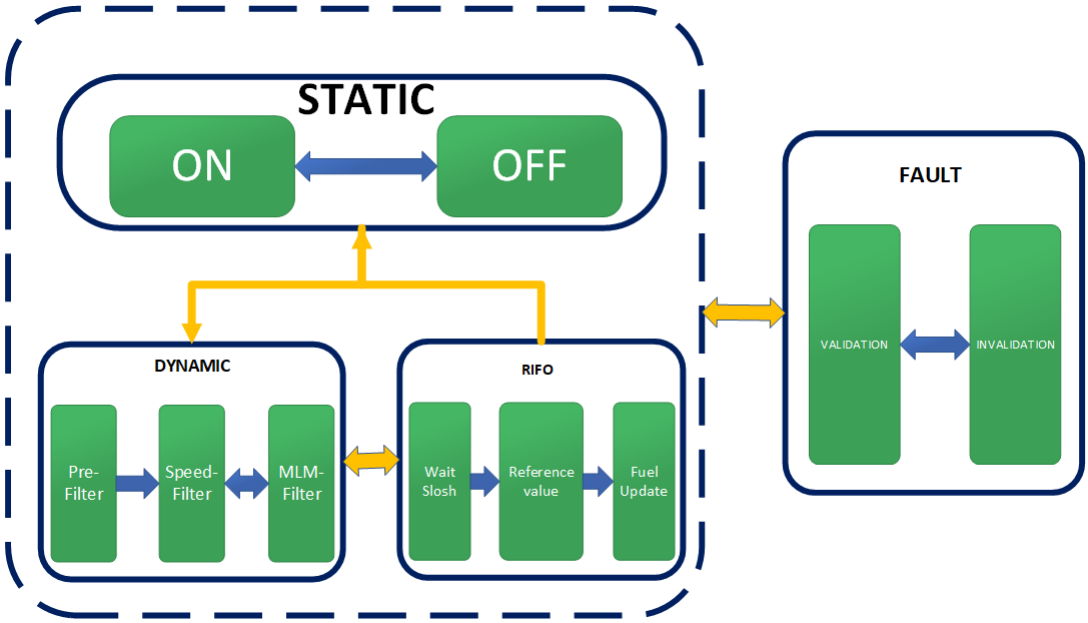


Figure 2.18: High-level representation of the whole model.

Chapter 3

Model Based Approach

Model Based Approach

The development of a well-suited environment for the automotive software architecture is not the only requirement in the application development process. As explained in the second chapter, the growing importance of the software has made a much quicker way for the vehicle function implementation necessary without forgetting essential features like efficiency and quality. Therefore, in order to be in line with the *"Faster, Better and Cheaper"* Pedro Rustan's motto, the need to come up with a new solution gave birth to an original methodology.

In comparison to the past, when *C-code* identified the one-of-a-kind approach for the development of an application, recently the *Model Based Software Design* has become the new popular alternative for modelling a system. In fact, the fall in number of bugs together with the automated code generation are just some of the advantages of this approach. However, from a wider view the *Model Based Software Design* is part of the *V-Model* diagram. This is a V in shape scheme that describes the specific steps to follow from the system requirements to the final release of an average application. According to the figure 3.1, while the left hand side deals with the 'implementation phase', the rising part concerns the 'validation phase' that will be analyzed in the fourth chapter.

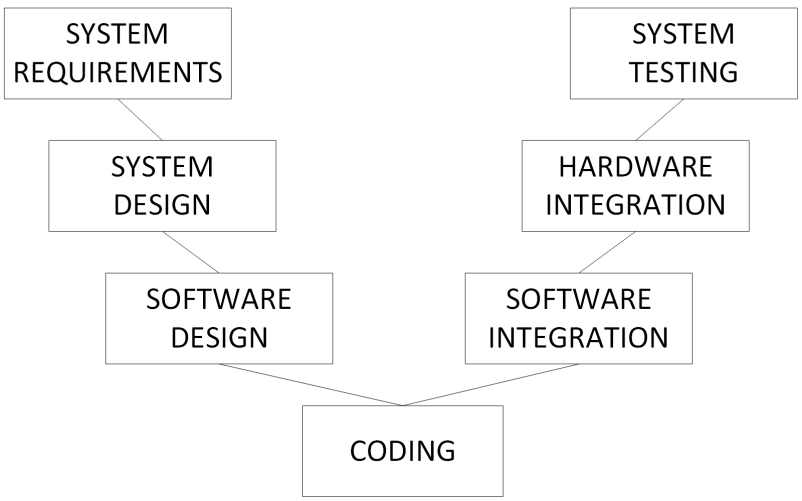


Figure 3.1: V-Model scheme.

3.1 First Half of the *V-Model*

Due to the first phases of the *V-Model* being strongly related to the description of the application (figure 3.1), any mistake or lack of information in the function requirements are prone to be discovered within this section. In the case of a distributed system that exchanges data with other applications, an error could spread over the whole structure and damage the system.

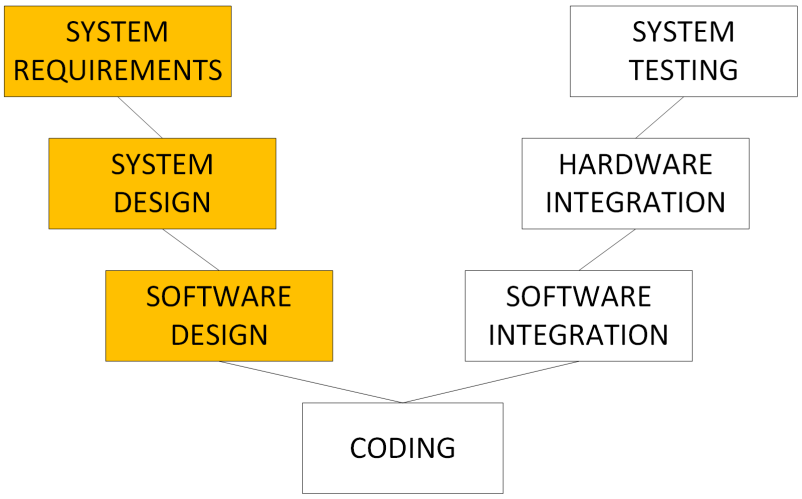


Figure 3.2: Verification Phase.

3.1.1 System Requirements

Albeit *System Requirements* seems to be a new concept, it was indirectly mentioned in the section 2.4. In particular, it concerns the exhaustive, clear and unambiguous description of the functionality. In fact, incomplete requirements provoke gaps that must be filled by developer's experience. Whereas, unclear information is the origin of a bad design and, consequently, wrong data exchange with other functions. Finally, ambiguous explanation leads to results that differ from the intended purposes.

Regarding the *Fuel Level Control*, the whole function logic is illustrated by three files:

I/O File : the main goal of this file is to provide details about type of function (triggered or periodical), inputs, outputs and local variable to set.

Logic File : explains how to manage data of the vehicle function, the expressions associated to the filters and the events to move from one state to the other. Hence, it includes the algorithm.

VF File : gives further information about the logic and the number or type of interconnected vehicle functions.

3.1.2 System Design

Similarly to the *System Requirements*, the System Design has already been seen in the second chapter. In particular, it covers the high level design of the vehicle function without any software tool. In this phase, there is a first passage from the theory to a high-level modelling of the vehicle function where the main transitions and sub-systems are highlighted.

Therefore, the goal of this stage perfectly matches with the section 3.3 where the theory of the *System Requirements* was sketched in order to achieve a first prospective of the vehicle function. Obviously, the outcome of the System Design is not definitive. In fact, if some information is overlooked or misunderstood, the model will undergo further changes later.

3.1.3 System Description

In comparison to the first two steps, the *System Description* describes how the implementation is performed from a software point of view and, consequently, offers an alternative to the the one-and-only *C-code* approach. Although hand-written programs enable developers to achieve a higher optimization in terms of data type, variable and allocating memory, the number of disadvantages is significant.

The most evident drawbacks are time-consuming delays to trace back the origin of a bug, difficulties to test single function of the entire program and issues to integrate new technologies. As a result, it is undeniable that the consequence of all these effects is a tardiness in the realization of the final vehicle function. However, in order to overcome these obstacles and fasten the software development, the *Model Based Software Design* was devised.

Overall, this is a visual method that simplifies the description of an average mechatronic system through the design of models and interconnections. The environment, within the system is modelled, is a software with detailed features. In comparison to the hand-written *C-code*, this approach enables programmers to test single sub-systems, to generate code autonomously (C, VHDL, C++, etc.) and to modify the model instantaneously. Consequently, it introduces a middle layer between logic description and code generation that eases the test and implementation of the whole model. Thereby, the application and physical target become for the first time two distinct entities that cooperate once the code is loaded on the embedded device.

However, a further difference between the *C-code* and *Model Based Software Design* regards the way whereby the test of functionality is performed. The *V-model* takes into consideration four different procedures for testing:

1. *Model in the loop*: test is performed in the simulation tool and enables to check the behaviour of the program during the development stage. If the

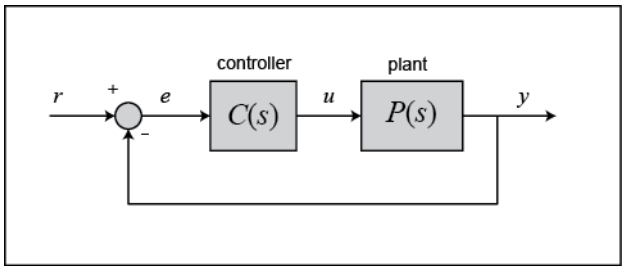


Figure 3.3: Controller and Plant

function worked properly, it is likely that the test on the hardware platform would not have difficulty during the integration.

2. *Software in the loop*: part of the model belongs to the simulation tool and part is in executable C-code.

3. *Processor in the loop*: in comparison to the *Software in the loop*, the executable C-code runs on a specific hardware.

4. *Hardware in the loop*: part of the code runs in a real time simulator, and the other is on a physical hardware.

While the *C-code* requires loading the entire code before detecting possible mistakes, the developer immediately checks whether the implementation of a new model behaves correctly in the *Model Based Software Design*. However, the advantages of the the latter come at a price. In fact, although the graphical description results easier to understand and analyze, not only does an efficient code generation require specific libraries but also the length of the program is higher than handwritten C-code and, consequently, takes a longer time to be executed.

3.2 StateFlow

For the *Fuel Level Control* implementation, the choice of the *Simulink* software has been made. This is a graphical block diagram tool developed by *Mathworks* that provides a standard library with blocks for: logic operations, mathematical expressions, discrete or continuous transfer function, inputs/outputs ports and programming cycles such as *if-else*, *for*, *while* ans so on. Overall, a block is associated to a functionality that receives inputs by means of connectors and could return one or more outputs.

Within the *Simulink* environment, a model can also include external toolbox blocks and a *StateFlow* chart that plays the essential role in representing sequential decisions, like a finite state machine, via states, transitions, junctions and graphical functions. Regarding the scheme of the vehicle function, it involves a *StateFlow* chart, that carries out the *Fuel Level Control* logic, and *Simulink* blocks to simulate inputs coming from the bus and the outputs to forward. In addition to this, it is evident that data can be either local to the *StateFlow* chart

or shared with external blocks.

3.2.1 States

A state represents an operating mode of a sequential system whose activity/inactivity depends upon the transition that allows the logic to reach the state. If the condition is verified, the transition can be performed otherwise the logic is stuck in the same state. However, according to the hierarchical relation in the chart, a state can be defined as *superstate* or *sub-state*. It easy to understand that their names are related to the location of the states. In fact, while the former clusters one or several sub-states, the latter is part of a superstate. For example, the figure 3.4 shows a superstate *A* and two sub-states *B* and *C*. It is worthwhile to underline that if there is a transition to a sub-state, the activation of the B or C states implies that its superstate A will be active as well.

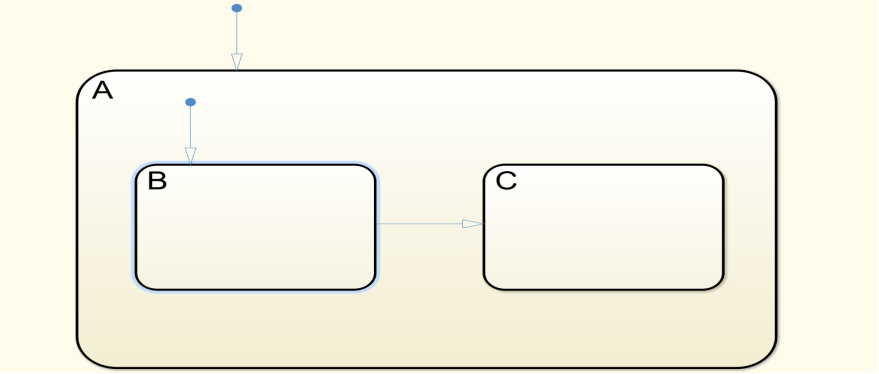


Figure 3.4: State example.

State Action

Regarding the content of a state, it tends to comprise functions to call, variables to set or events to trigger. However, the time-line of the action is scheduled according to the type of *state action*. There are five actions:

Entry Action: the action is performed as soon as the state becomes active. It is preceded by the word *entry* or simply *en*.

During Action: the action is performed until the state is active. It is preceded by the word *during* or simply *du*.

Exit Action: when the condition of the output transition is verified, the action is performed before the logic moves to another state. It is preceded by the word *exit* or simply *ex*.

On Event Action: if the event happens, the action is executed. It is preceded by the word *on NAMEVENT*.

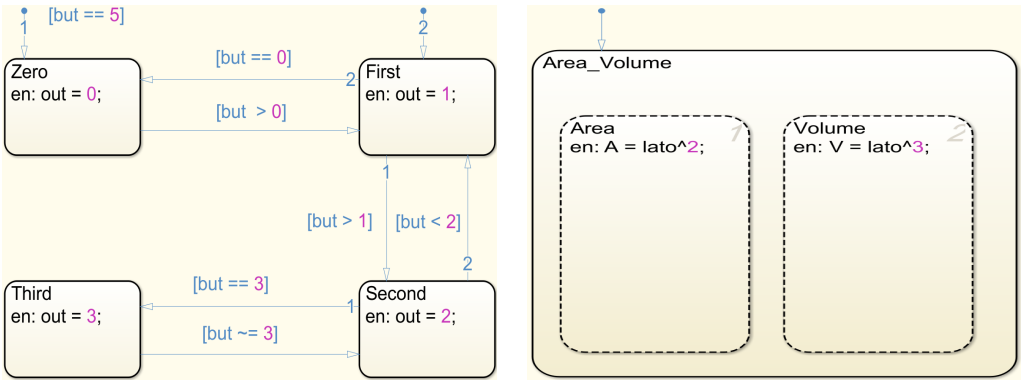
Bind Action: is used to bound a variable to a specific state. In that case, the variable cannot be modified in another part of the chart.

Decomposition

Although at the beginning of the 'State' paragraph was outlined the bond between the activity of a state and the associated transition, this is just one of the possible conditions. In fact, the execution order of the states is related to the system that is going to be developed.

For instance, the lift dynamics in a block of flats with three floors identifies a sequential system that can be modelled as a *StateFlow* chart with three states (one for each floor) whose transitions are controlled by the lift buttons as in figure 3.5(a). It is obvious that in order to reach the desired destination, all the middle floors must be passed through. From a developing view, this means that during the execution, there will be consecutive transitions from the origin to the destination taking into consideration all the states in-between. In this case, the floors are *exclusive* states because only one of them can be active at any time instant and they are figured in solid rectangles.

On the other hand, if the goal of the *Simulink* model is to figure out the area and volume of a square, the chart can be implemented through *parallel* states because area and volume are independent (figure 3.5(b)). Consequently, they can be computed at the same instant once the side length of the square is received. Although a *parallel* decomposition guarantees the execution of more states simultaneously, the number on the top of the state assigns the specific order to follow. The reason behind this choice is linked to the code generation phase. In fact, this enables to describe parallel states via if-else conditions rather than multiple tasks.



(a) *Lift implementation.* **(b)** *Area and Volume implementation*

Figure 3.5: *Parallel* and *exclusive* sub-state examples.

3.2.2 Transitions

The transition is a line that connects two states. In the case of a transition without condition, this is performed immediately. Therefore, a *during action* in the origin state should be useless because the logic moves directly toward another state. Whereas, in a transition with condition, the passage occurs if and only if the condition is verified.

On the whole, a transition is described through the *Transition Label*. It does not include just logical conditions but follows the structure below:

$$[condition]\{condition\_action\}/transition\_action \tag{3.1}$$

- *Condition*: is a logical statement with *or*, *and* and *not equal* symbols and is written in square brackets.
- *Condition Action*: is enclosed in curly braces and describes an action to perform such as increment a counter as soon as the *condition* is verified.
- *Transition Action*: refers to a function or an event to trigger.

However, if there are several transitions to reach the same state, the developer will set an order of priority. In order to demonstrate the benefits of transitions, the figure 3.6 shows the *Lift* implementation from another point view without writing any action inside the states:

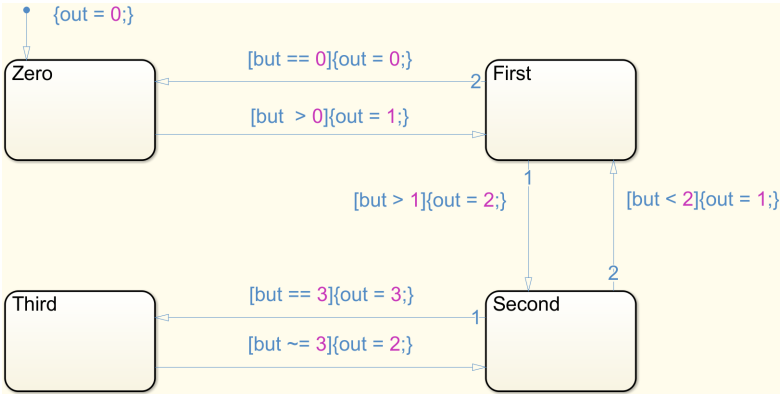


Figure 3.6: *Lift* implementation through transitions.

Default Transition

According to both the figures 3.5(a) and 3.5(b), it is clearly evident the presence of the two arrows without origin and destination to one of the states of the charts (*Area_Volume* and *Zero*). This is called *Default Transition* and is the entry point of the chart. Whatever is the value of the inputs, the first step of the logic is the state connected to the *Default Transition*. Albeit there could be multiple default transitions in the same chart, the treatment is exactly the same of an average transition. This means that the developer lays down an execution order

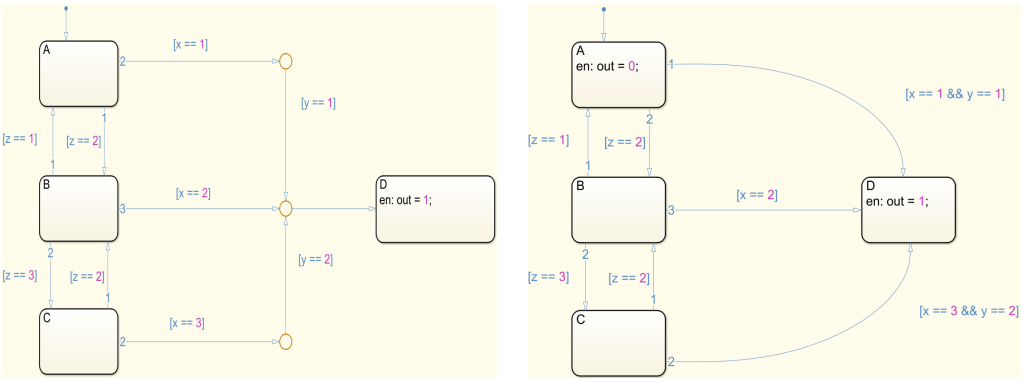
among them in order to avoid mistakes.

The default transition purpose shows the striking contrast between the *lift* and *Area_Volume* examples. In fact, while the latter contains two *parallel* sub-states that don't require a *default transition*, this is necessary for *exclusive* sub-states otherwise the logic of the superstate could not be carried out.

3.2.3 Connective Junctions

Although the *connective junction* is not truly a milestone of the *StateFlow* approach, it is particularly useful for testing. In fact, rather than writing several conditions on the same transition, junctions enable to divide the line in as many segments as the number of conditions. In this way, if a transition is not performed, the developer detects immediately where the false condition is.

However, connective junctions can be employed also to simplify the model. When multiple transitions have the same destination, they could be divided in two: the segments for the condition and the shared link for the destination. As depicted in figure 3.7, junctions have the role in linking condition segments coming from different states to the same destination. It is interesting to note that the condition segment of the state A and C is divided in two parts in order to check whether both conditions are verified during the execution of the chart.



(a) *Transition with connective junction.* **(b)** *Transition without connective junction.*

Figure 3.7: The use of connective junction.

3.2.4 Graphical Function

The *graphical function* is an element that defines a function through connective junctions and transitions. It is a modular and reusable object which is defined in a separate window in comparison to the chart and is called inside states or transitions. Essentially, a *graphical function* consists of a default transition and two junctions which are the entry and end point of the function.

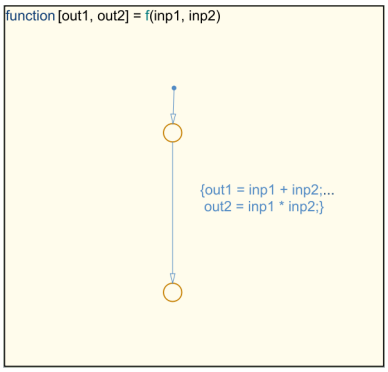


Figure 3.8: Graphical Function example.

Concerning inputs and outputs, they are declared in definition. For instance, the expression below defines a function f with two inputs (inp1, inp2) and two outputs (out1, out2). Whereas, the figure 3.8 shows a possible implementation.

$$[out1, out2] = f(inp1, inp2) \tag{3.2}$$

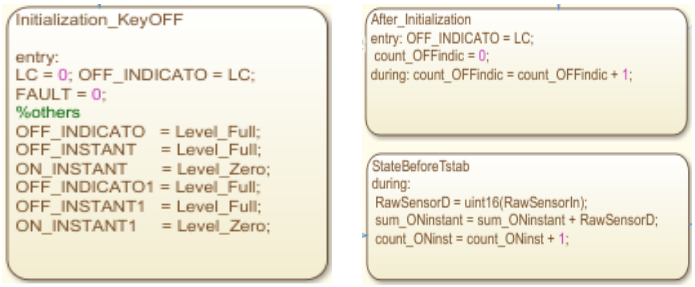
3.3 Vehicle Function Implementation

According to the *StateFlow* rules and the logic of the vehicel function, the whole model of *Fuel Level Control* function can be split in two superstate: *KEY_ON* and *KEY_OFF*. While the latter deals with the initial variable setting and the sampling of the fuel level when the engine is turned off, the former includes all the working conditions except for the FAULT. It is important to stress the concept that some details of the implementation will be hidden for company reasons.

3.3.1 STATIC

As mentioned in the 2.4.3 section, the STATIC has to guarantee a minimum sampling of the fuel level and output a reliable percentage to the instrument panel. Due to these events being independent functions, they can be modelled with two *parallel* sub-states (*OnInstantComputation* and *REFUELING_CHECK*) that share just the *lock* variable (figure 3.11). This local data aims to make the fuel level available as soon as the initial sampling is finished. In this way, the output is not forwarded within that interval.

However, a further situation to bear in mind is the transition to the *KEY_OFF* state during the initialization sampling. In that case, the sampling of the fuel sensor is not interrupted but continues in a sub-state of the *KEY_OFF* and all the required variables for the STATIC computation are update and ready to be used later. Concerning graphical functions, they are employed in the logic of the STATIC not only to compare fuel sensor data but also to verify the transition among superstates.



(a) *Inizialization state.* **(b)** *AfterInitialization state.*

Figure 3.9: OFF state view.

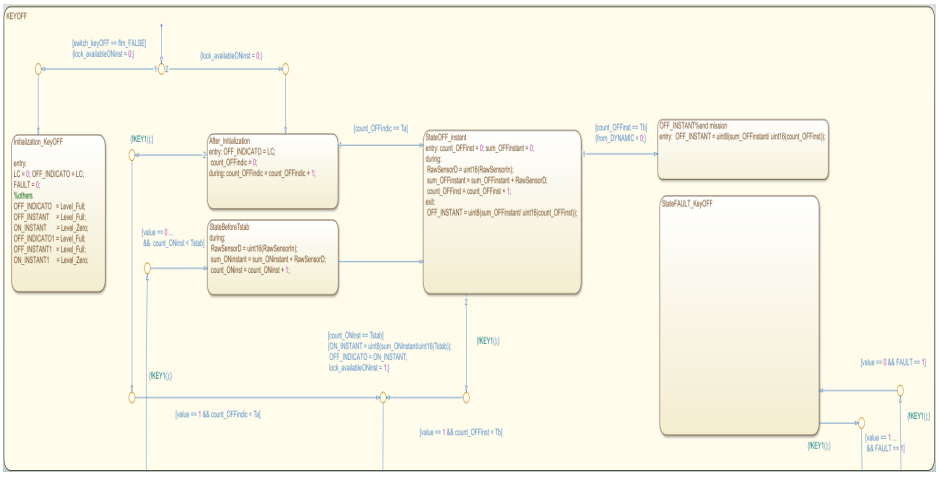
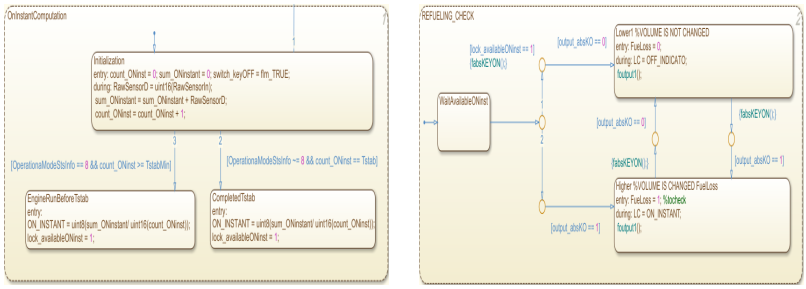


Figure 3.10: OFF state.



(a) *OnInstantComputation state.* **(b)** *RefeuelCheck state.*

Figure 3.11: STATIC view.

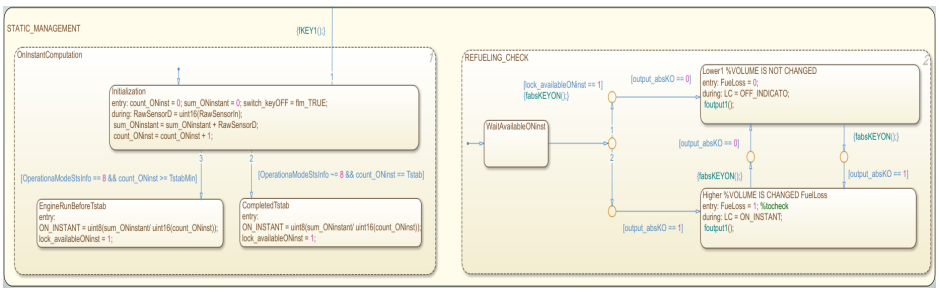


Figure 3.12: *STATIC* state.

3.3.2 DYNAMIC

The typical events that provoke a transition from the STATIC to the DYNAMIC could be both the engine starting or a really quick turning on and off of the vehicle. In the latter case, the STATIC management is not taken into consideration and the logic moves directly to the DYNAMIC. However, from an implementation perspective, there are two transitions with the same origin and destination. Therefore the use of a connective junction perfectly suits the vehicle function needs.

Regarding the body of the DYNAMICS, this is made up with two *exclusive* sub-states. As long as the the fuel level is higher than *MLM_min*, it is computed in the *Normal Dynamics* that exploits the graphical function properties to develop the dynamics of the *Pre-Filter* and the *Speed-Filter*. Whereas, when the MLM strategy becomes active, the *MLM-Filter* is implemented by means of a graphical function that computes the fuel percentage. However, as soon as the fuel level exceeds the *MLM_max* threshold, the logic goes back to the *Normal Dynamics*.

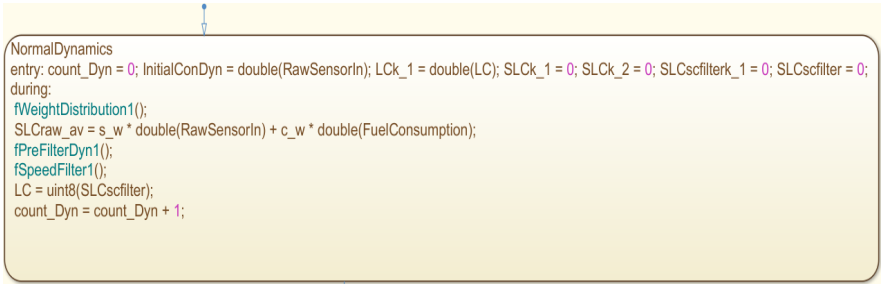


Figure 3.13: *NormalDynamics* state.

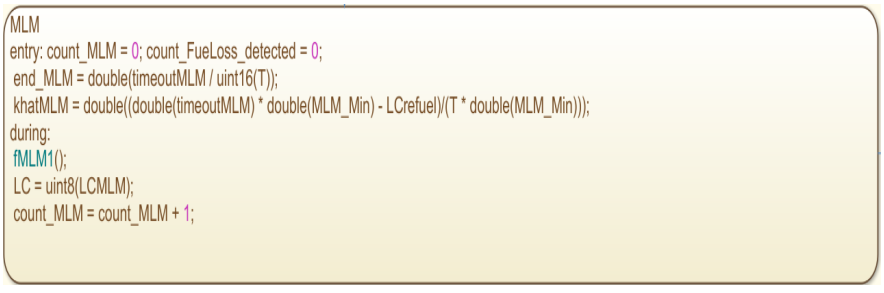


Figure 3.14: *MLM* state.

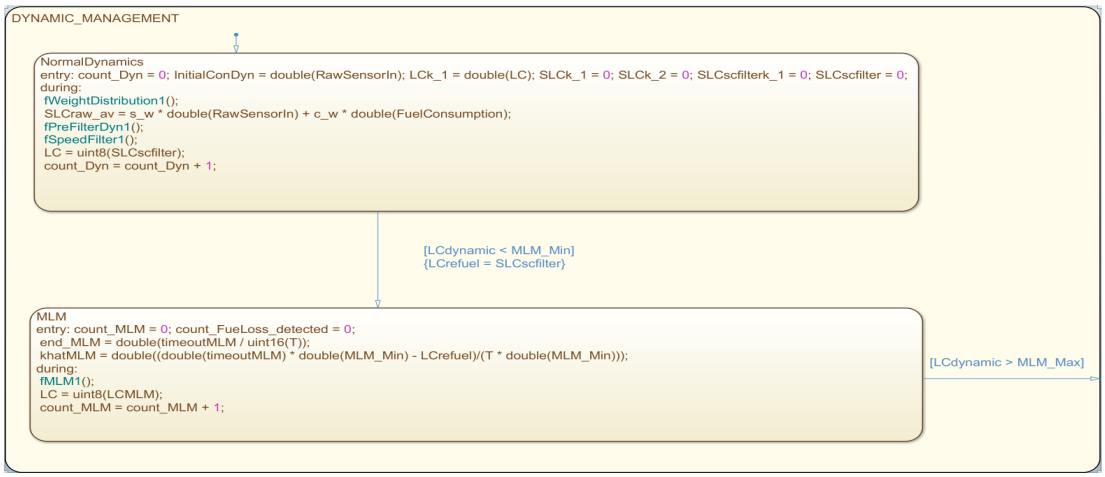
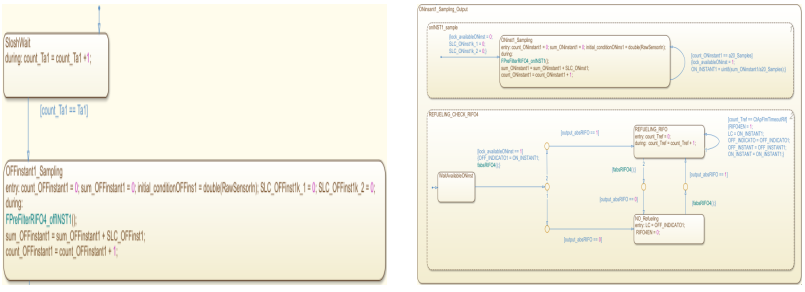


Figure 3.15: *DYNAMIC* state.

3.3.3 RIFO

The whole structure of the RIFO condition can be divided into three sequential conditions. At the beginning, the dead time to attenuate the fuel slosh effect is represented by a state which contains a counter that is incremented until the transition to move in the consecutive state is true. Secondly, a reference value is figured out through the mathematical average of the only fuel sensor samples. The aim of the reference value is to define a landmark that will be compared to an updated measurements of the fuel level. In this way, the driver is constantly aware of the fuel level variation. For the sake of the knowledge, all the data acquired by the sensors are subjected to the *Pre-Filter* dynamics for cleaning reason.

On the whole, the development consists of three *sequential* sub-states: *SloshWait*, *OFFinstant1_Sampling* and *ONinsant1_Sampling_Output* (figure 3.16). However, what is noticeable from the *ONinst1_Sampling* sub-state is that there is a transition whose origin and destination is identical. This is called *Inner Transition* and, in our case, it is used to keep the logic up-to-date with the current fuel level. Henceforward, there could be two possible events. The refuel action is finished but due to the car being turned on with vehicle speed equal to zero, the logic remains in the RIFO state (figure 3.17). Otherwise, the vehicle accelerates and the logic moves to the DYNAMIC.



(a) *OnInstantComputation state.* **(b)** *RefeuelCheck state.*

Figure 3.16: *Parallel* and *exclusive* sub-state examples.

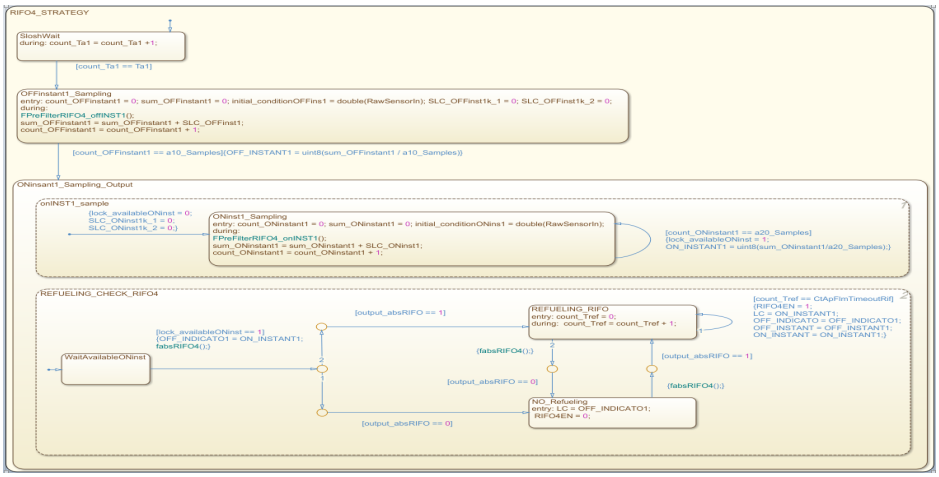


Figure 3.17: *RIFO* state.

3.3.4 OUTPUT

The aim of the OUTPUT is to manage the fuel lamp according to the quantity of fuel in the vehicle. Because this functionality is required in both the DYNAMIC and RIFO equally, an efficient implementation involves the *Refuel_Dynamics_OUTPUT* super-state with two *parallel* sub-states. One for the OUTPUT and the other includes the RIFO and DYNAMIC *exclusive* sub-states. This allows not only to avoid redundant functions but also to light the fuel spy regardless of which working condition is running.

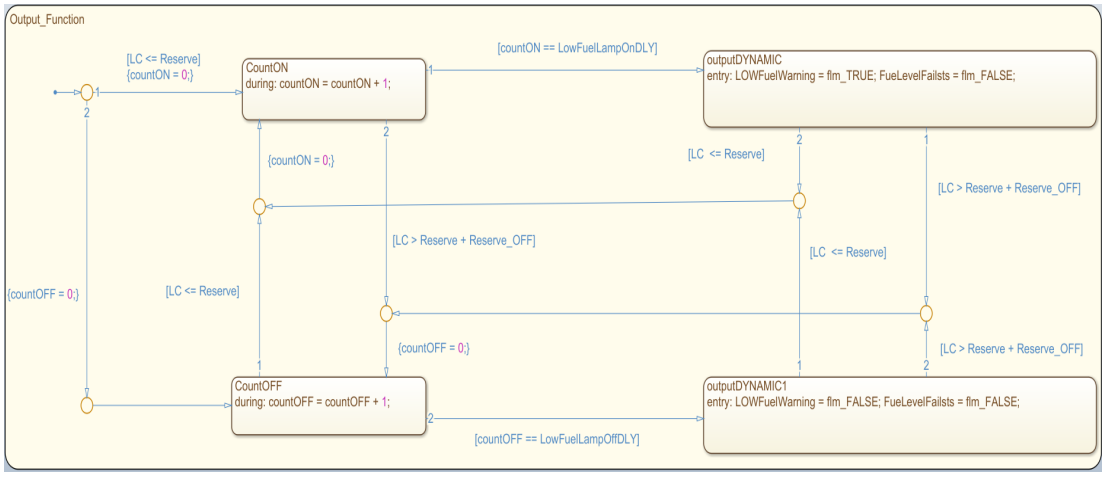


Figure 3.18: *OUTPUT* state.

3.3.5 FAULT

The FAULT state is the alternative to the ordinary vehicle function operation. In case of fuel sensor data mistakes for a certain period, the fault is validated and the output is set to zero percentage. This enables the driver to detect the malfunction and do car maintenance because the error is displayed on the body

control until it is present.

From the implementation perspective, before reaching the *FaultOutput*, there are two preliminary states that validates or cancel the fault. Inside the *FaultOutput* state (figure 3.18), the fuel level is set to zero and fuel warning deactivated. However, if the *Key-Off* condition occurs while the error is confirmed, the algorithms needs to be robust enough to avoid accidental error cancellation. Therefore, the logic moves in a specific state of the KEYOFF state until the issue is fixed.

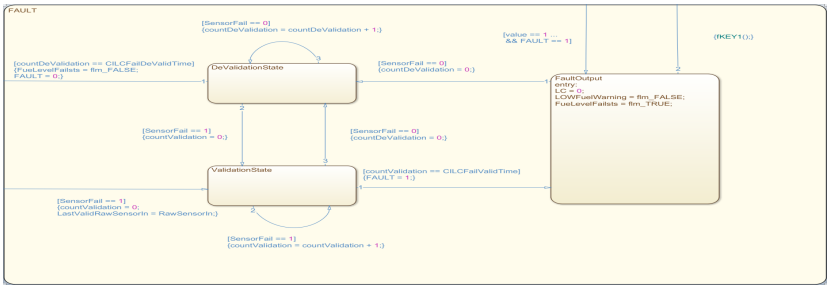


Figure 3.19: *FAULT* state.

3.4 Vehicle Function Model

Similarly to the final part of the second chapter, this section tries to give an overall view of the implementation. For a clearer comprehension of the whole scheme, the sub-charts serve to cluster each single state (figure 3.20). From here on, the fourth chapter will focus on the *Validation* area of the *V-model*.

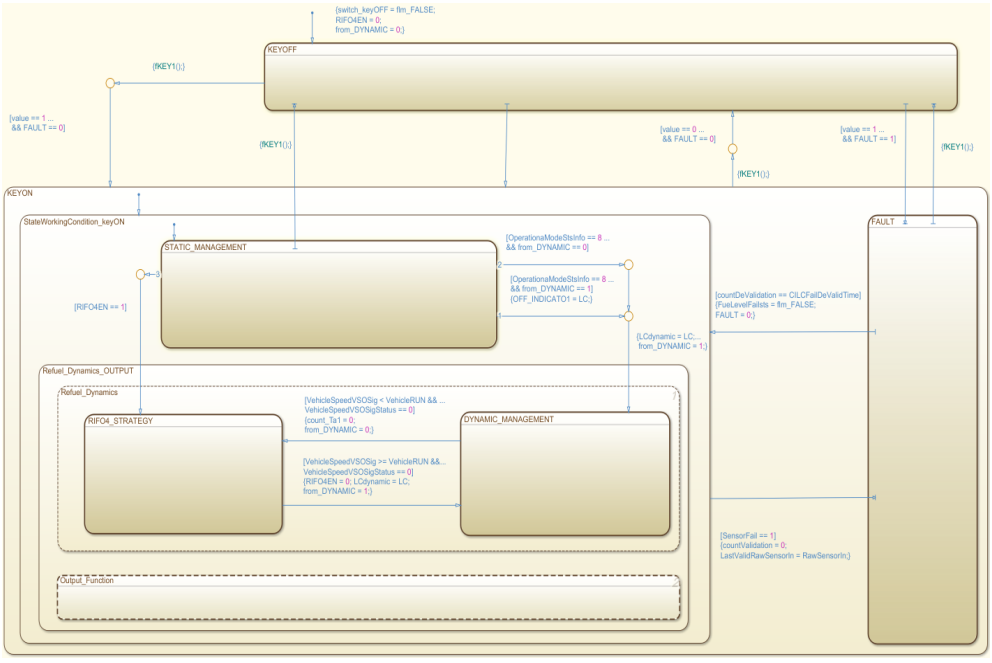


Figure 3.20: Vehicle function implementation.

Chapter 4

Code Generation and Integration

Code Generation and Integration

As introduced in the *Model Based Approach* chapter, one of the most significant advantages of the *V-Model* has been the opportunity to outline the necessary stages to follow for the development of a software application from the original idea to the test on real-time devices. Although the *V-Model* purpose is clear, so far the analysis has covered just the initial phases.

Since the end of the *Software Design* phase, the development process of a vehicle function experiences a different approach regarding testing and integration. Henceforth, the *StateFlow* model is the starting point of the *'Validation Phase'* that consists of four stages: *Code Integration*, *Software Integration*, *Hardware Integration* and *Acceptance Test* (figure 4.1). The final goal is to provide a reliable and tested application ready to be loaded and employed on a real-time system.

Although each phase of *'Verification Phase'* will be discussed in depth in the body of the chapter, the choice of not presenting the other side of the *V-Model* in the previous chapter has been made to pay enough attention to both the model based approach and the integration test without causing a lot of confusion to the reader. Therefore, dealing with these milestones of the thesis on two separate areas guarantees to cover details that the use of just one chapter would forget. However, it is unquestionable that *'Validation'* and *'Verification'* phase are two sides of the same coin.

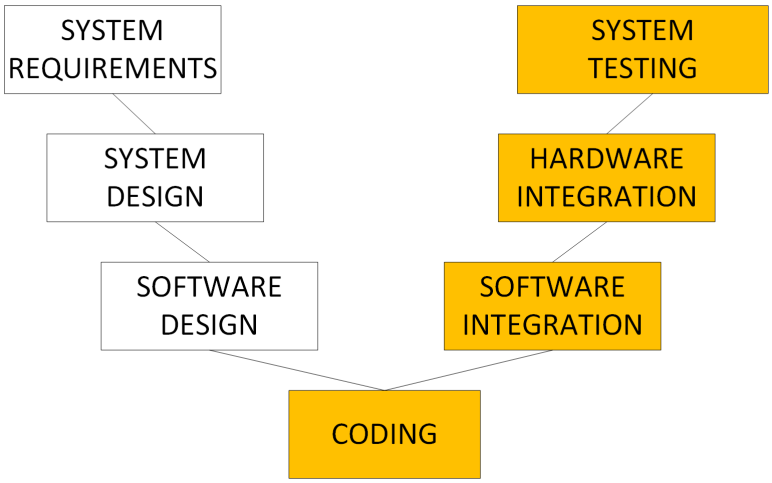


Figure 4.1: Validation Phase.

4.1 Coding

Similarly to the *Software Design*, the *Coding* represents the other outstanding revolution brought on by the *Model Based* approach. While the graphical sketch of the application makes the logic appear clearer to the developer's eye, the *Coding* phase introduces a new perspective about the *Code Generation* concept. Throughout the years, this task has been often associated to the compiler whose goal has been to turn source code in machine code.

Albeit *Code Generation* allows unskillful developers to generate C codes and companies to fasten the software development procedure, the choice of the software for the auto-code generation is not trivial because data types, variables definition and function calls have to be determined accurately. Therefore, it is undeniable that *Code Generation* is a double edged sword whose benefits depend upon the software features. Obviously, in order to achieve marginal differences between a generated and a handwritten code, the program needs to be as optimized as possible and easy to integrate into the system.

4.1.1 TargetLink

TargetLink is a transformation tool developed by *dSpace* that generates C-code directly from the *Matlab\Simulink* environment. Although the decision about the generator was influenced by company preferences and academic licenses, the reasons behind this choice regarded mostly the great benefits in terms of code efficiency and reliability. In fact, *TargetLink* guarantees test mechanisms, compliance with standards and specific data dictionary that make developers detect any kind of mistakes earlier and help programmers to manage blocks more efficiently.

However, the real bedrocks of *TargetLink* can be summarized in three areas: code, control design and Standard Support. In fact, the most important objective of a software generator is to represent a valid alternative to handwritten code. This implies that reducing time-consuming and error-prone issue have to be tackled and overcome easily. Thereby, a more efficient and quicker results are fulfilled. In order to see the forest for the trees, the solution proposed by the generator needs to be not only fast but also efficient and, possibly, cheap.

Within this context, the properties of the generated code appear to be crucial for the final development. *TargetLink overflow detection* allows the programmer to find the exact line of code where the error occurs. In this way, the mistake can be fixed rapidly. In addition, *TargetLink* offers an *Auto-Scaling* tool that enables the developers to save a large amount of time. It is worthwhile to define the 'scaling' process as a procedure to set the range of a variable and, consequently, the type conversion. Whereas, *Code Optimization* (figure 4.2) enables the generation of a reliable ANSI (*American National Standards Institute*) C-code with a slight difference in comparison to the handwritten code. The latter is a necessary requirement for a generator because a weak code could misuse embedded system

architecture through superfluous lines of code and useless variable declarations.

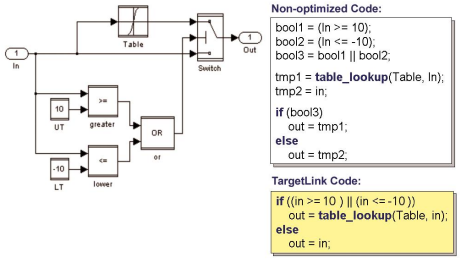


Figure 4.2: Block optimization.

Furthermore, *TargetLink* enhances the *Matlab* environment by means of specific blocks which extends the *Simulink* library. This facilitates a quick integration on the vehicle function model and a more efficient code generation because of the direct link between *TargetLink* blocks and generation tool. All in all, all the *Control Design* features are classified in the following list:

- Supported Simulink Blocks
- StateFlow support
- TargetLink Simulation blocks
- AUTOSAR blocks

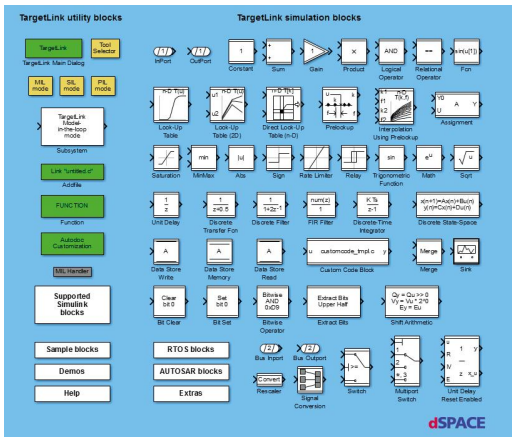


Figure 4.3: *Targetlink* blocks.

Regarding the relationship with the AUTOSAR standard, *TargetLink* offers a set of blocks that allow developers to model application accurately. As depicted in the figure 4.3, even though some blocks can be employed out of the automotive goals, the others are mandatory for the application description. For instance, the *Runnable* block serves to generate code in a separate C function. Whereas, the *Inports* and the *Outports* defines the interactions of the vehicle function with other ASW in compliance with the AUTOSAR standard. In our vehicle function,

the figure 4.4 shows how inputs and outputs of the *Fuel Level Control* function are modelled by means of the *TargetLink* blocks. While the logic of chart represented by the *StateFlow* chart remains unchanged, the *TargetLink* blocks take the place of the previous *Simulink* blocks.

Similarly to the AUTOSAR *Conformance Test Agency*, the conformity of *TargetLink* with international criteria are attested by the *TUV SUD*. This is a German association that certifies whether *TargetLink* is suitable for software development on safety-critical systems according to worldwide standards.

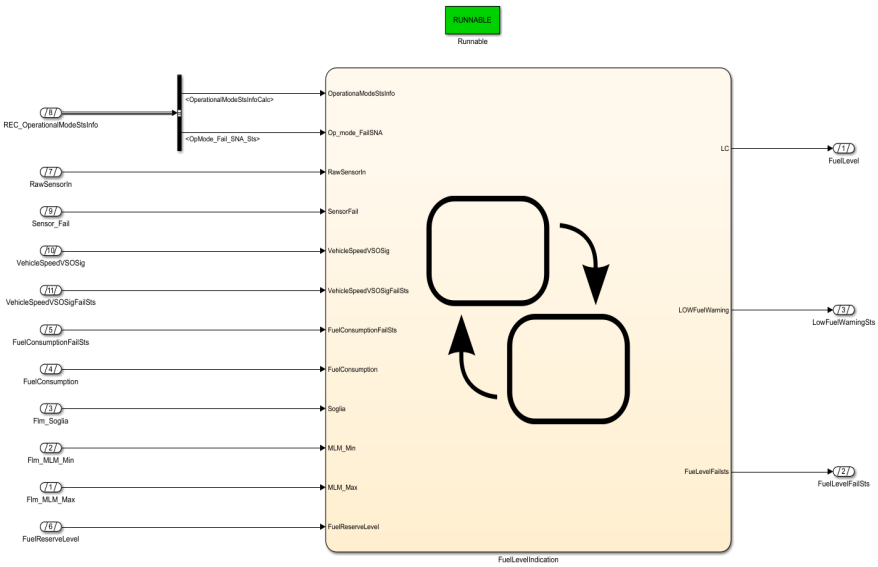


Figure 4.4: TargetLink block in the *Fuel Level Control* function.

4.2 Software and Hardware Integration

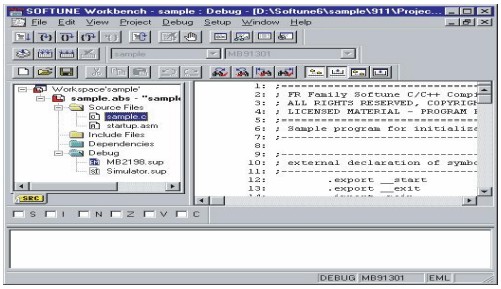
At the end of the coding phase, a source (.c) and header (.h) files are ready to be integrated in the system. However, the precise steps to follow rely on both the hardware target and the interaction of the vehicle function with other functions. Due to the *Fuel Level Control* requires data coming from other components, there has to be a check that certifies the right connection among the vehicle functions in order to compile the code correctly. This means that if the developer makes a mistake in the modelling phase, it will be forwarded in the following stages and the software that aims to link and verify the VF interconnections will return an error. As long as the error is present, the process is blocked here.

However, a further situation to take into consideration occurs when all interconnections are well-established but the VF doesn't work as the developer expects or, alternatively, the results differ from the simulation tests. This is a trickier event because the origin of the mistake is unknown. In fact, it could come from either hardware devices or the data management. In this case, a debugger together with a tool able to show transmitted data allow programmers and testers to detect the mistake and to monitor each single step of the logic.

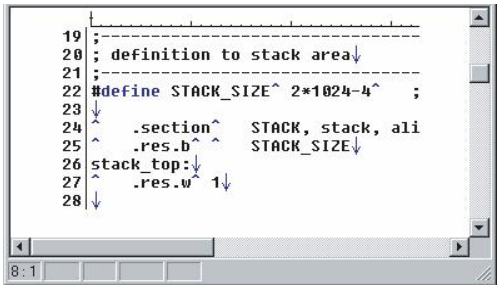
4.2.1 SOFTUNE Workbench

The *SOFTUNE Workbench* is an integrated development environment realized by *Fujitsu* which aims to compile and debug the generated code and brings together three kinds of debuggers and language tools such as compiler, assembler and linkage kit. Overall, this program can be split in three parts: *manager*, *debuggers* and *body*. While the former deals with to code and make programs, the choice of the debugger is made between simulator debugger, emulator debugger and monitor debugger.

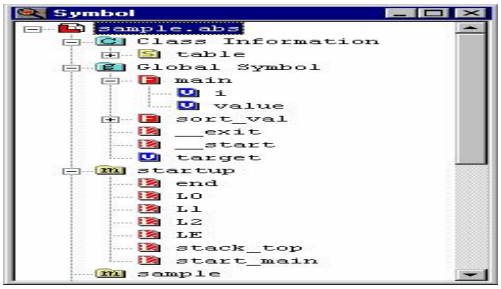
Before describing the use of this software in the thesis, it is worth underlining the structure of the main windows. Once clicked the icon of the program, the first window is the *Main Window* which is divided into three areas. The top section is covered by the *Tool Bar* that allows a developer to build, make and compile the program. The middle area consists of two sub-windows: the *Project* window and the *Edit* window, figure 4.5(a) and 4.5(b) respectively. While the former displays the location of the program and its structure in terms of files and folders, the latter makes the developers interact with the running code. Finally, the bottom part (figure 4.5(d)) depicts the *Output* windows which informs the user about the result returned by the program.



(a) *ALL.*



(b) *Edit.*



(c) *Project window.*

(d) *outptut window.*

Figure 4.5: Workbench environment.

In this project, the use of *SOFTUNE Workbench* concerns both the hardware and software integration. Initially, it has been employed to compile the generated code and, later, to build the whole code on the hardware device. Before running

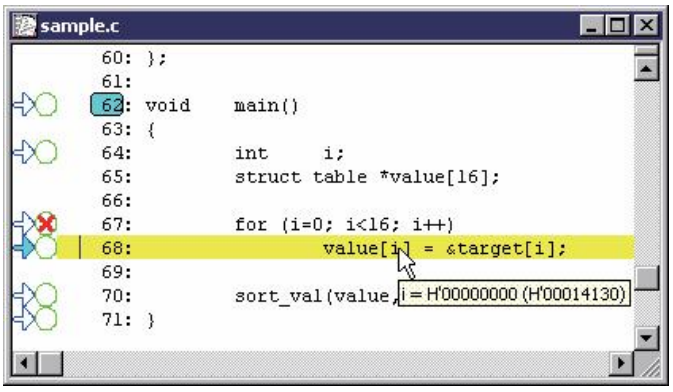


Figure 4.6: Break-point window.

the code on the ECU, there is a middle stage that confirms the interconnections of the *Fuel Level Control* and links the header file to the project. This stage is performed by the *make* functions that returns a target file ready to be loaded on the ECU. In this situation, if the *output* windows shows a 'no-error' message, the entire code can be flashed on the device definitely.

On the other hand, unexpected mistakes could appear during the integration of the code on the hardware device. For this reason, *SOFTUNE Workbench* offers both a *breakpoint* mechanism (figure 4.6) that enables to stop the code and to show the value of a variable during the execution. However, it is important to note that there is not a direct communication between *SOFTUNE Workbench* and ECU. In fact, a hardware device has to be employed in the chain to establish a correct data exchange. It is called emulator and is used not only to make *SOFTUNE Workbench* debug the code but also to load the whole project on the ECU.

4.3 Acceptance Test

In order to deliver a secure and functional application, the testing phase is the final part of the *V-Model* diagram. Although the *Model Based Design* allows developers to perform tests during the *Software Design* stage, the bench test on the hardware target aims both to act as a counter-check for the modelling tests and to be the conclusive proof before the commercial use of the application. Therefore, it is undeniable that test cases need to be worked out precisely and to take into consideration all feasible situations that could occur in reality.

From a company point of view, the Testing team has the role to come up with test cases for each vehicle function and, in case of unexpected results, send back the errors to the Software Development team. Obviously, in this project, test cases have the purpose to verify the accuracy and correctness of the *Fuel Level Control* vehicle function without forgetting that they were not devised by a team.

Figure 4.7: CANcase device.

4.3.1 CANAnalyser and CANCase

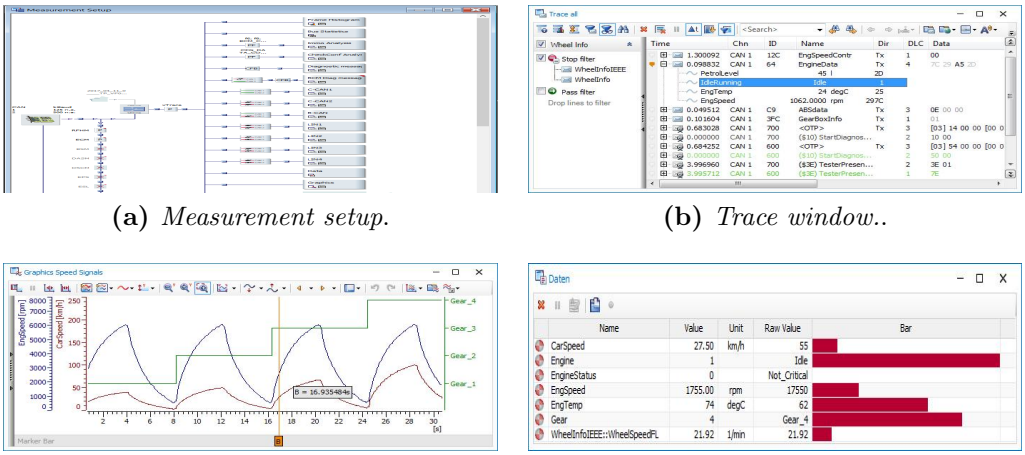
Regardless of the type of test, data coming from other vehicle functions have to be simulated in order to verify all the interconnections of the vehicle function. For this reason, a software able to show which information are floating on the communication network together with a device that connects this software to a hardware simulator must be employed in the bench tests.

The *CANcase* is a hardware device made by *VECTOR* that works as an interface between *CANanalyzer* and the hardware simulator which aims to simulate data exchange on the CAN and LIN network (figure 4.7). Obviously, the channel combination depends upon the device features and the number of output ports. In our case, due to the fact that the LIN was not required, two ports for the CAN-H and CAN-L were enough for the bench test.

Concerning the software, *CANanalyzer* is a tool that is utilized for the network analysis. From a graphical viewpoint, its intuitive structure enables the user to handle data clearly and efficiently. It consists of a several windows that allow the tester to monitor the data exchange, to plot the outputs that are forwarded on the CAN and, finally, to change at run-time the data coming from other vehicle functions. However, the type of testing could be performed either via a script or, simply, changing data manually.

The figure 4.8(a) shows the *Measurement Setup* where a graphical representation of the data flow is displayed. More precisely, while the right hand-side shows details about the communication like BUS statistics and graphics, the bottom part enables to disregard unnecessary node and to specify the type of simulation[1]. Whereas, the *Trace* window displays the type of message and its content (figure 4.8(b)). Finally, the *Graphic* window plots the values of the signals on the graph (figure 4.8(c)) while the *Data* windows shows value, unit and name of the signal (figure 4.8(d)).

[1]There are two kinds of simulation: P indicates a script whereas IG is immediate simulation.



(a) *Measurement setup.* **(b)** *Trace window..*

(c) *Graphical window.* **(d)** *Data window.*

Figure 4.8: CANanalyzer environment.

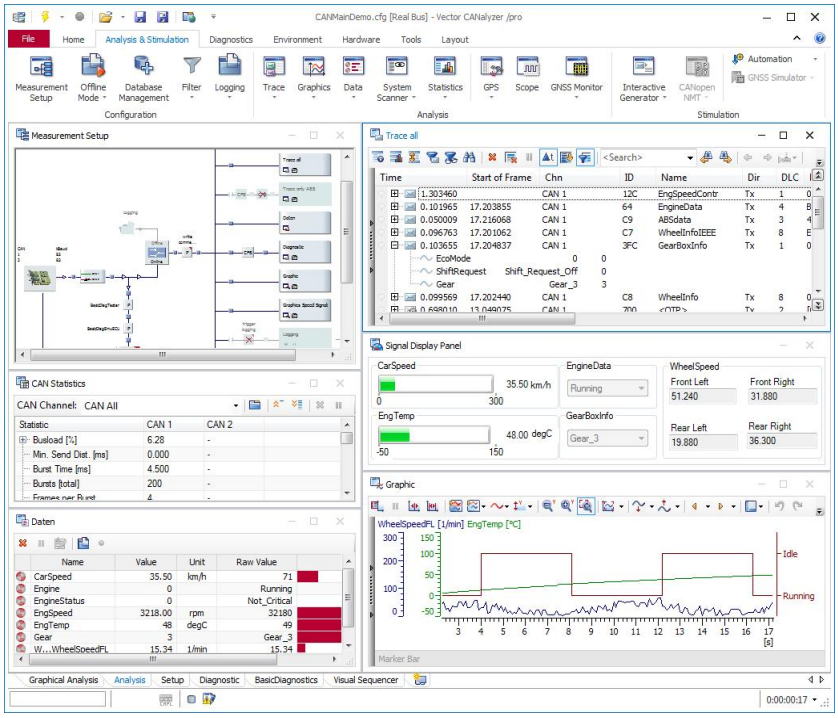


Figure 4.9: CANanalyzer image

4.3.2 Hardware Simulator

Although the *CANanalyzer* and the *CANcase* are mandatory to simulate data coming from other vehicle functions, these products cannot represent sensor signals. Therefore, the use of a hardware simulator is necessary to provide fuel sensor measurements to the ECU. Overall, a simulator can be described as large metal rectangular box that is made up with pins, levers, variable resistances and displays whose objective is to behave as much as close to real sensors in a vehicle.

As mentioned before, the connection between *CANcase* and simulator makes *CANanalyzer* detect any change on the CAN and display the real behaviour with

slight delays. In this specific project, the fuel sensor data are simulated through two variable resistances that represent the sensors inside the fuel-tank. Rotating these resistances the data received on the ECU changes as well and, consequently, the logic will perform accordingly. The table 4.1 illustrates how variables are distributed and controlled between *CANanalyzer* and the hardware simulator, while the figure (TO INSERT) illustrates the interconnection between previously described softwares and hardware devices.

CANanalyzer	***HW Simulator***
Vehicle Speed	Fuel-Level Sensor Data
Vehicle Speed Fail	Fuel-Level Sensor fail
Operational Mode	
Fuel Consumption	
Fuel Consumption Fail	

Table 4.1: Variable distribution.

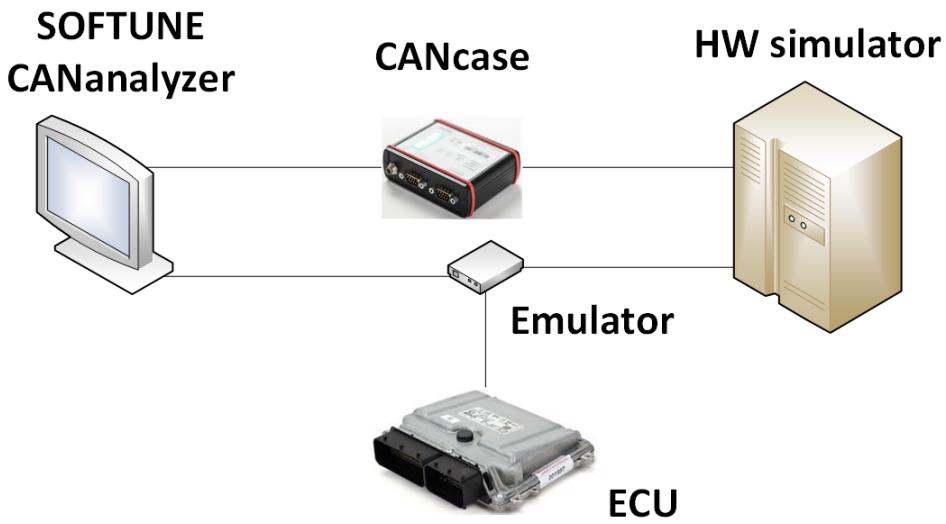


Figure 4.10: Sketch of the hardware and software interconnection for test bench.

4.4 The Final V-Model

In conclusion, taking into consideration the whole description of the first part of the *V-Model* together with the deep analysis provided in this chapter, the final *V-Model* diagram can be drawn as follows:

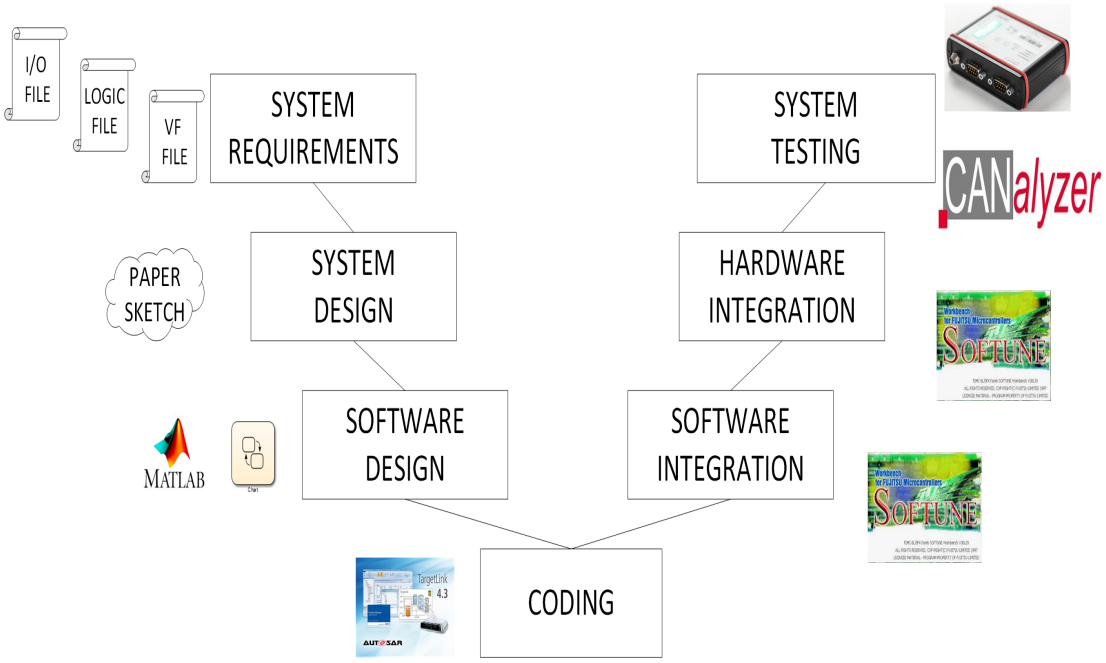


Figure 4.11: Final V-Model

Chapter 5

Test Cases

Test Cases

Before being delivered to the user, every kind of engineering applications have to overcome the testing phase. As mentioned earlier, the model based approach guarantees two types of test. While the modelling test occurs after the design of the vehicle function, the second and final tests is executed during the hardware in the loop phase on the hardware target. Although the environment is completely different, the results of the test cases should have marginal differences.

Regarding this project, the guidelines of the test cases were devised with the aim of outlining the strength of the application in common scenarios and finding out possible weakness in unusual situations. Therefore, the number of test cases is clustered in three areas: STATIC, DYNAMIC and RIFO. Finally, the last test case shows a complete automotive dynamics where all the previous results are taken into account. However, it is worthwhile to highlight that simulation and bench tests aim to verify the correctness and robustness of a vehicle function but their results are likely to differ from a real situation.

5.1 STATIC

5.1.1 Test Case 0

In this context, the car is turned on and the fuel level is 100%. Obviously, due to the vehicle not moving, the driver notes that the dashboard is lighted up and the fuel indicator is stable to the full fuel icon. However, it is noticeable a slight contrast between the figure 5.1 and 5.2 at the beginning of the graphs, the cause is a delay related to the hardware simulator that makes the software overlook the initial sampling time.

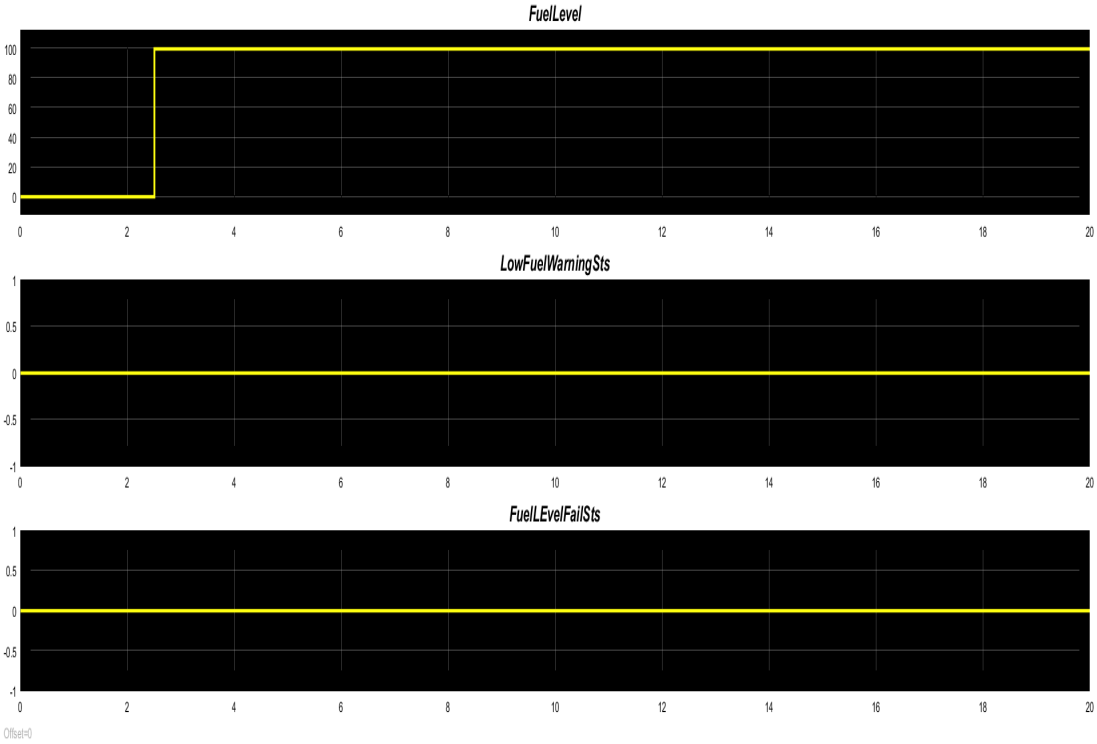
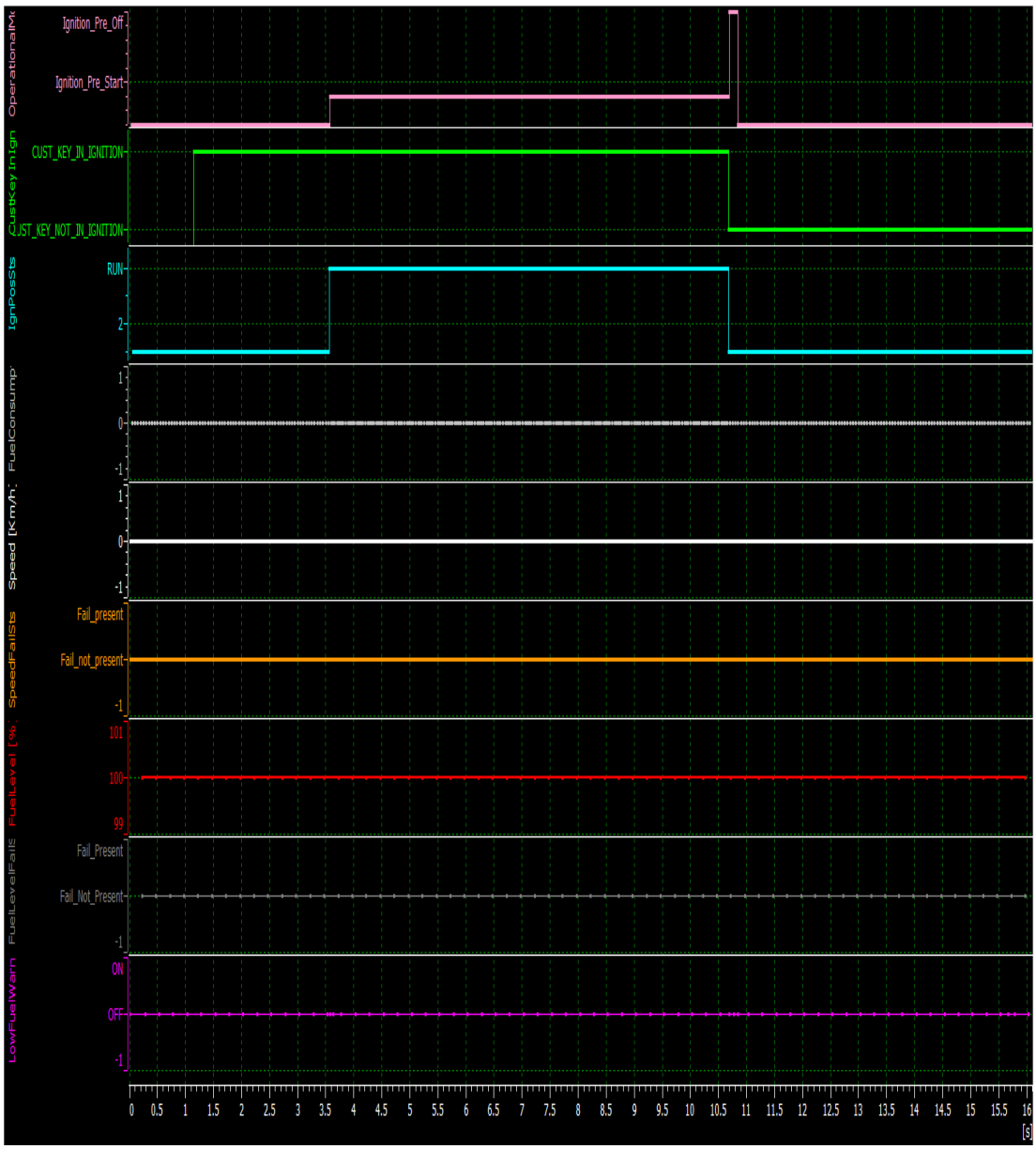
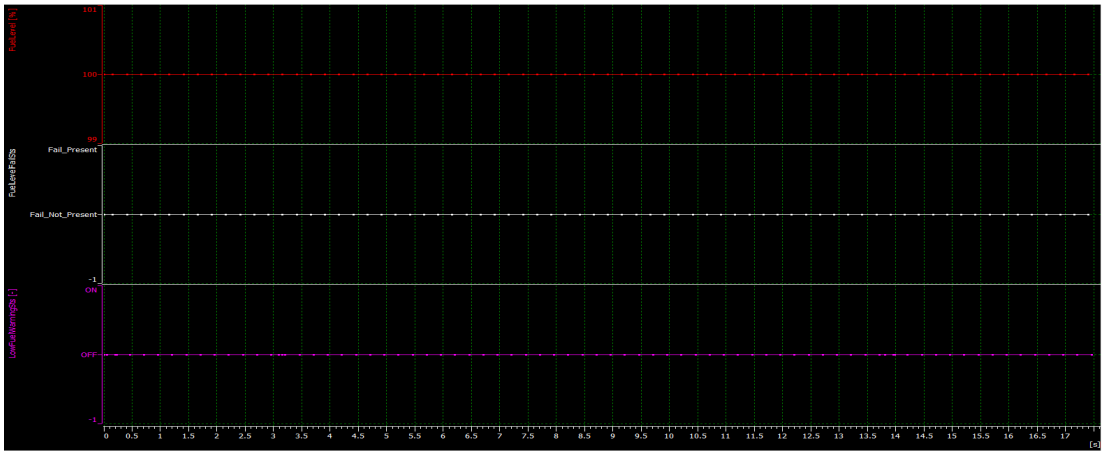


Figure 5.1: *Simulink* STATIC test0.



(a) *STATIC bench test0.*



(b) *Zoom of STATIC bench test0.*

Figure 5.2: Bench Test 0.

5.1.2 Test Case 1

Although the situation is pretty identical to the *Test Case 0*, the goal of this example is to verify that the lamp of the Low Fuel Warning turns on when the key is rotated. Due to the *reserve* parameter is set to 16%, the driver looking at the dashboard notes the lighted spy.

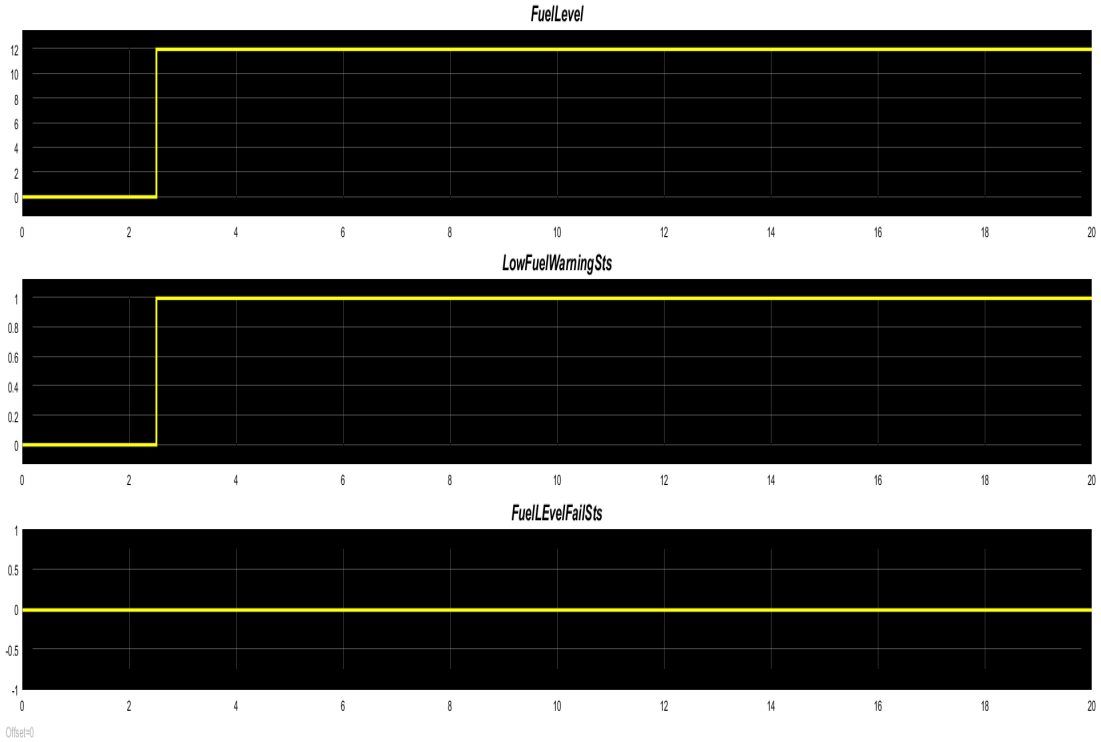
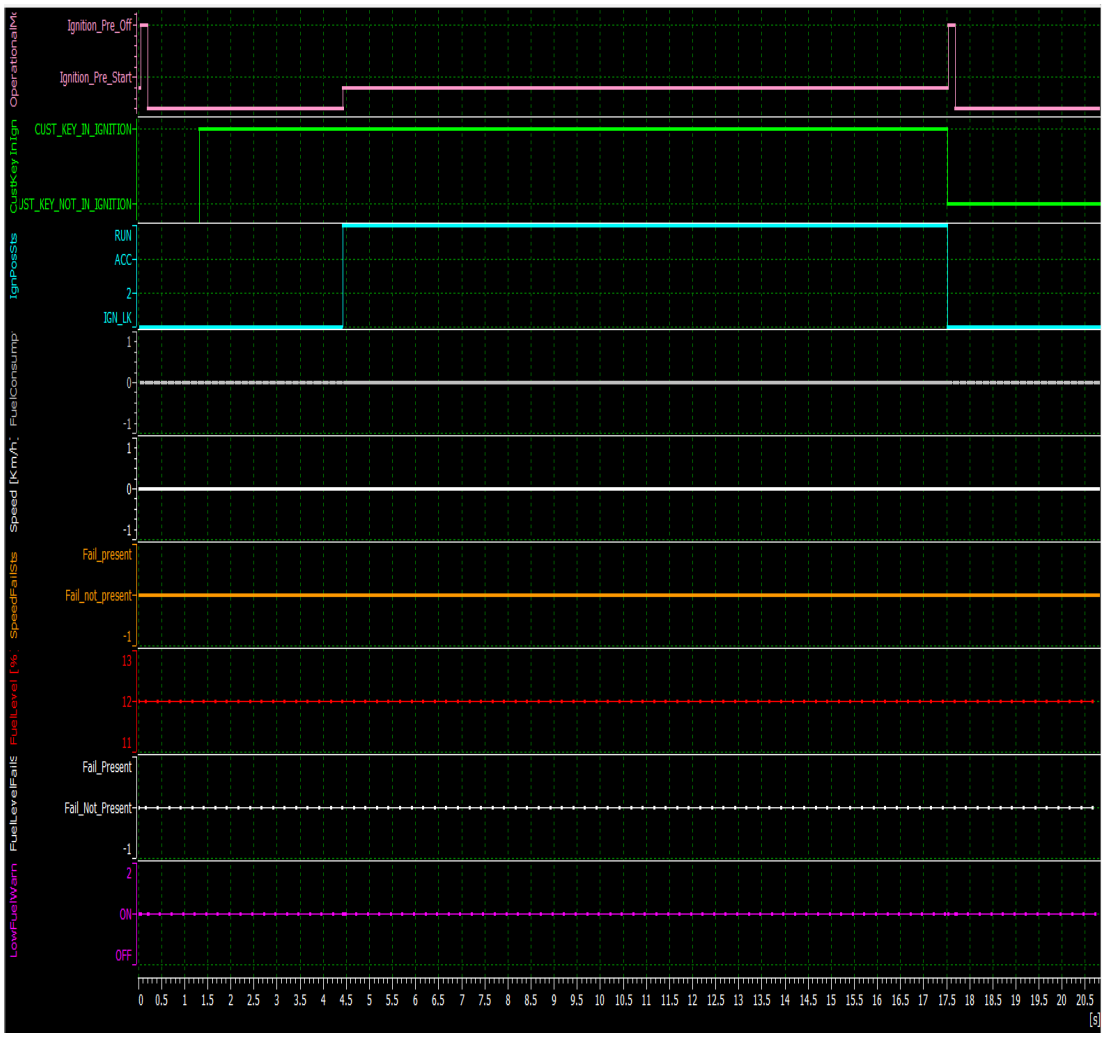
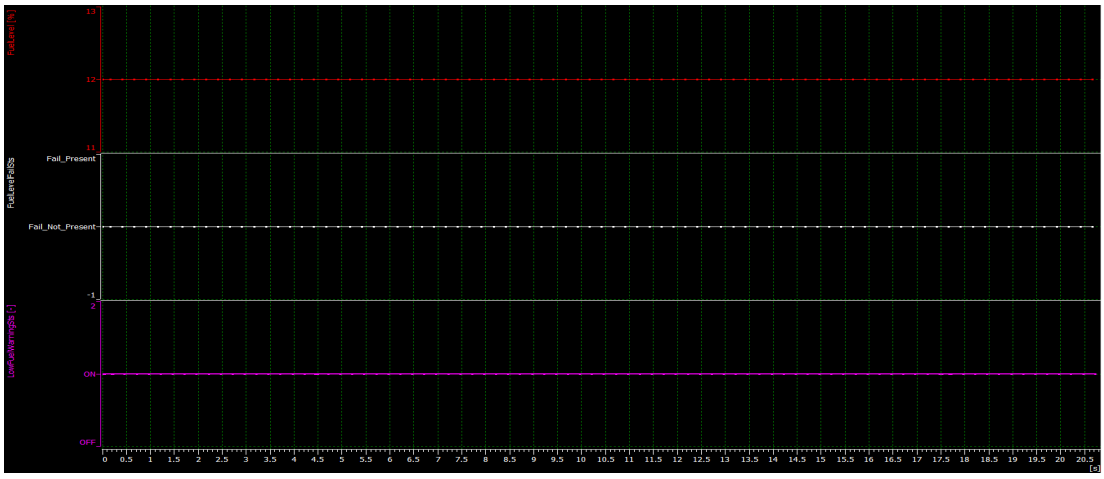


Figure 5.3: *Simulink* STATIC test1.



(a) *STATIC bench test1.*



(b) *Zoom of STATIC bench test1.*

Figure 5.4: Bench Test 1.

5.2 RIFO

5.2.1 Test Case 2

This condition is typical for an automobile which is stopped at the petrol station and is going to be refueled. The purpose of this test case is both to update the fuel level variation and to display the exact quantity of fuel inside the fuel-tank. The whole dynamics can be split in two situations: before and after refuelling. While the former experiences an initial percentage lower than the *reserve* threshold, the former ends with the fuel-tank completely full. In particular, as depicted in the figure 5.5 and 5.6, as soon as the fuel level surpasses the *reserve*, the lamp is turned off.

Albeit both *Simulink* and bench simulations don't have striking contrasts, it is interesting to outline that to simulate a fuel positive variation during the test bench, two knobs representing the variable resistances, have to be rotate slowly otherwise a significant and sudden step should be displayed during the simulation on *CANanalyzer*. Conversely, the model simulation requires just a line which describes the probable fuel variation.

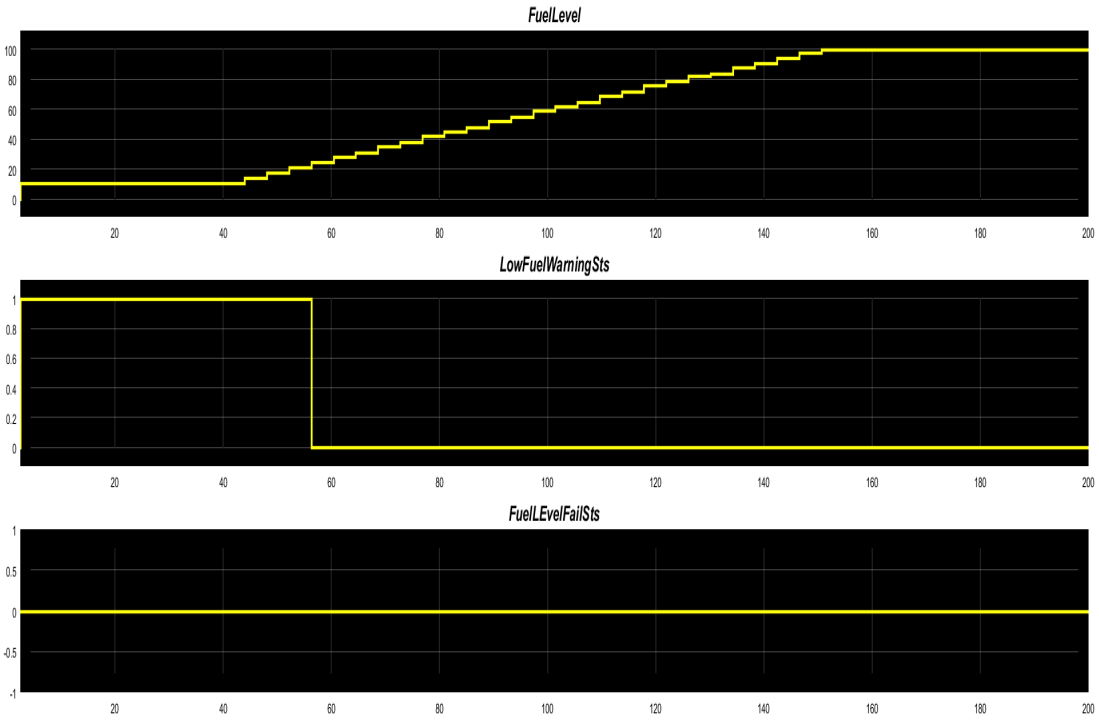
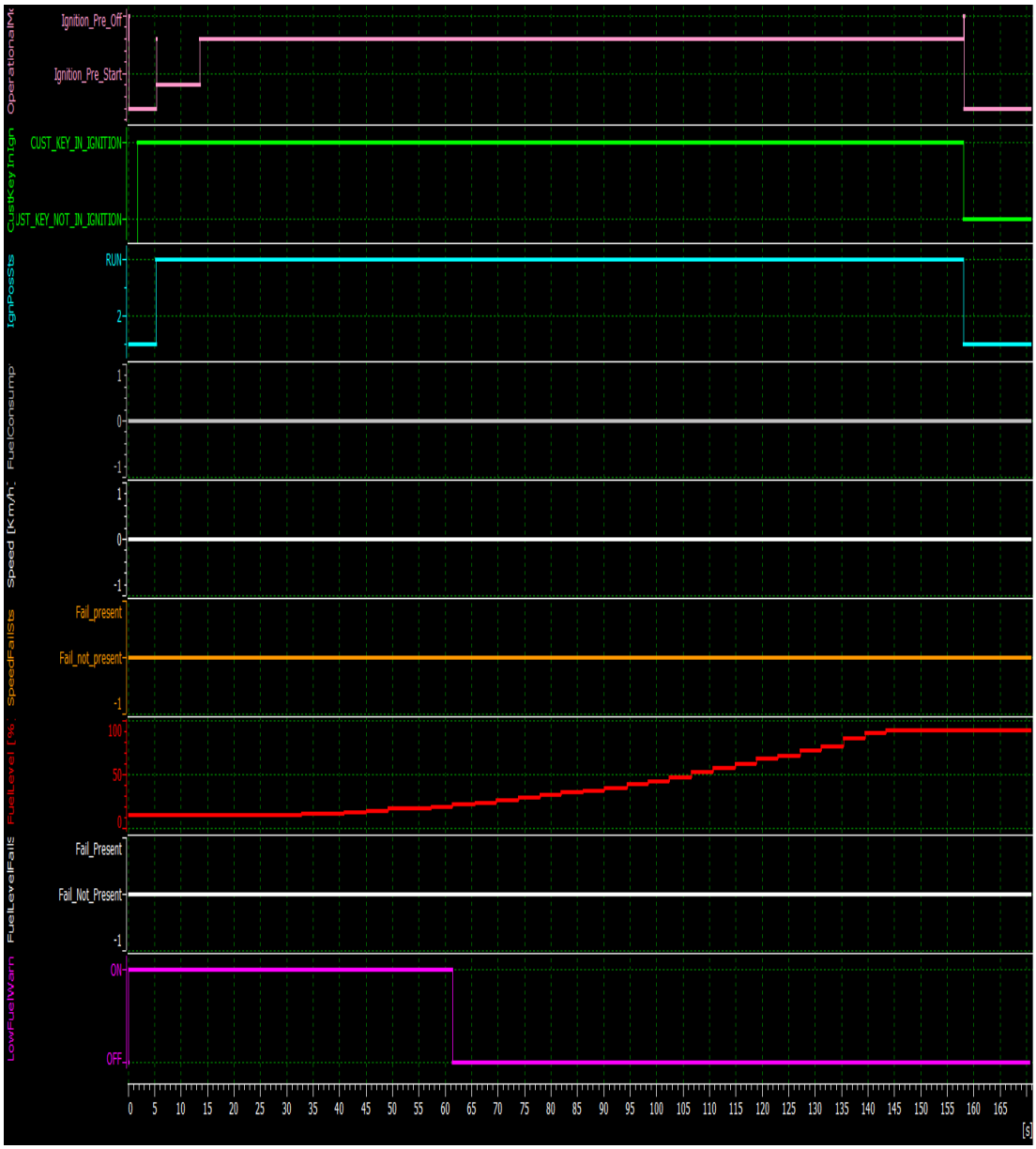
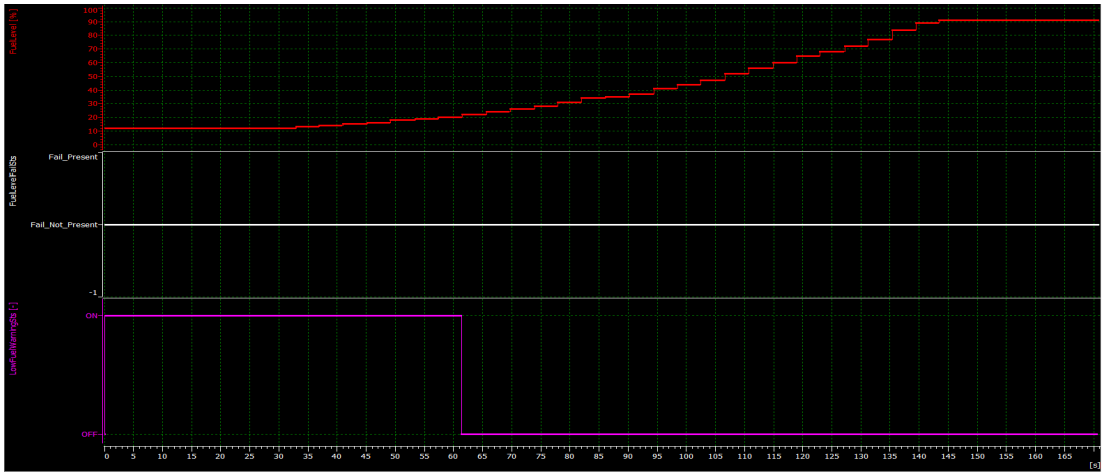


Figure 5.5: *Simulink* RIFO test2.



(a) *RIFO bench test2.*



(b) *Zoom of RIFO bench test2.*

Figure 5.6: Bench Test 2.

5.3 DYNAMIC

The DYNAMIC tests take into account all the events with the engine on. However, it is important to underline that the following test cases are based on two hypothesis. First, the *FuelConsumption* parameter is assumed to be zero. The reason behind this choice is essentially the difficulty in simulating an outcome coming from another vehicle function. In fact, because it relies on unknown vehicle features, carrying out a reliable estimation of the *FuelConsumption* resulted difficult to achieve and this would entail untrustworthy results.

Secondly, the fuel-tank is supposed to be completely full during the initial sampling and, suddenly, zero after the start up phase. For this reason, as soon as the transition to the DYNAMIC occurs, both the *Pre* and *Speed* filters detect empty fuel-tank and, consequently, the computed fuel level will go through a rapid decrease in comparison to everyday situations. Hence, the final decisions to disregard *FuelConsumption* and to set to 0% the fuel level after the start-up phase are made in order to observe the sole evolution during the DYNAMIC conditions. However, a real event guarantees a reliable *FuelConsumption* value, a more stable fuel measurements and, lastly, a more long-lasting simulation.

5.3.1 Test Case 3

The vehicle is accelerating at a constant speed of 27 km/h without fuel loss and the goal is to estimate the amount of time within the fuel percentage reaches zero. What is noticeable from the *Simulink* and bench test is that the car runs out all the fuel after approximately 2800 and 2500 seconds, respectively.

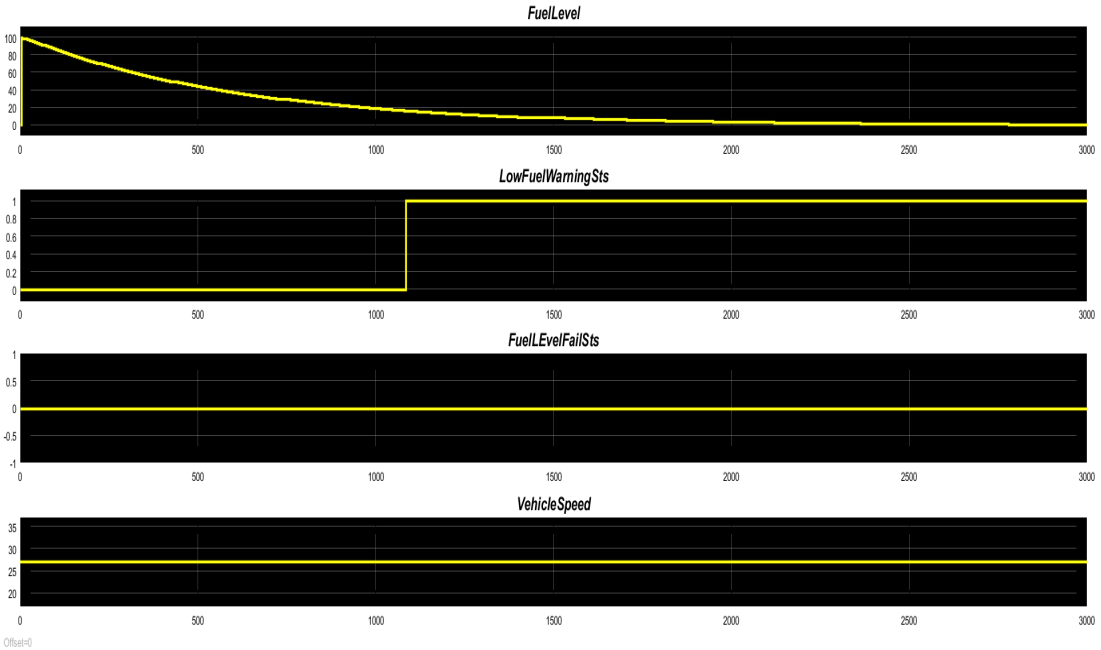
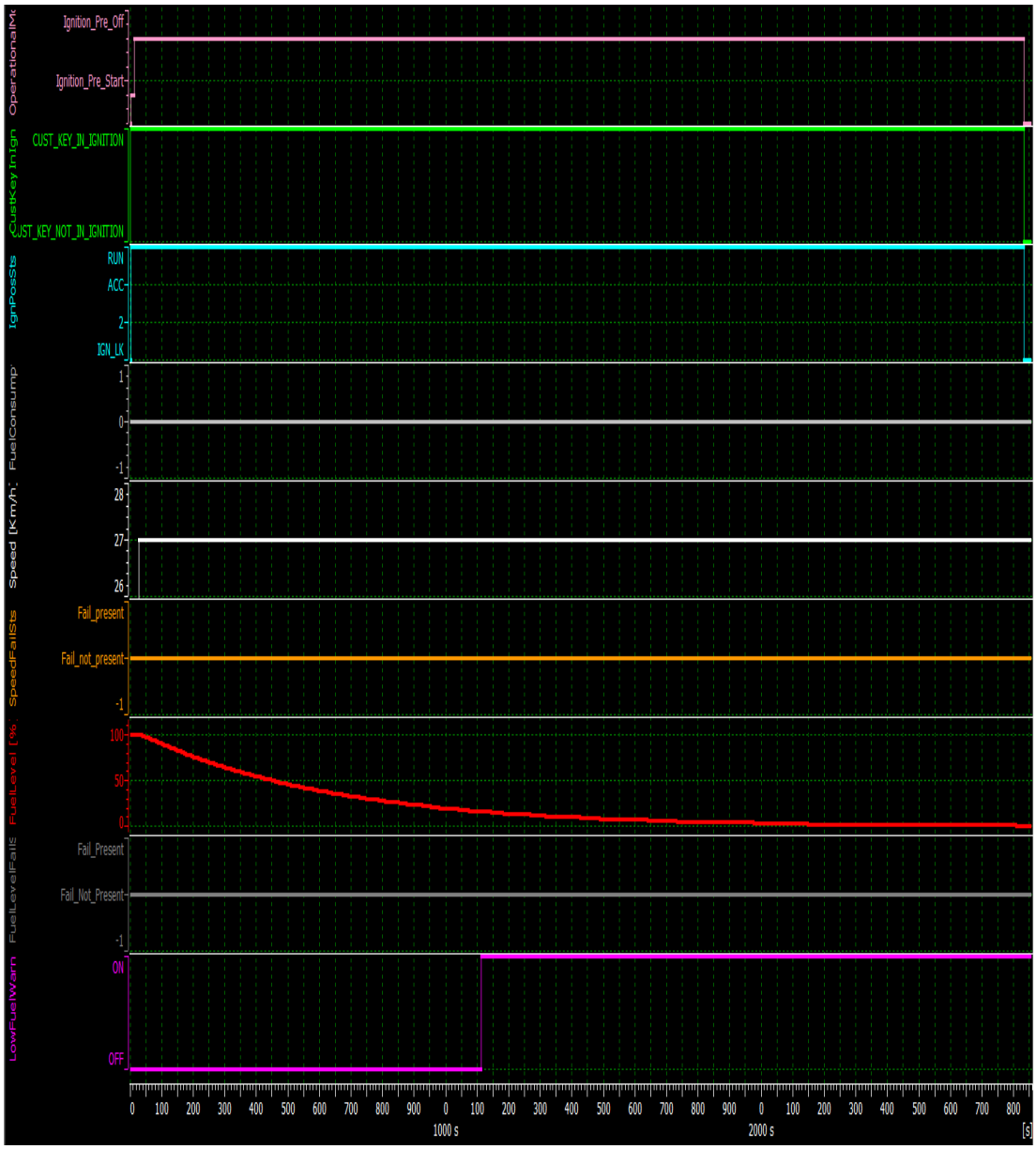
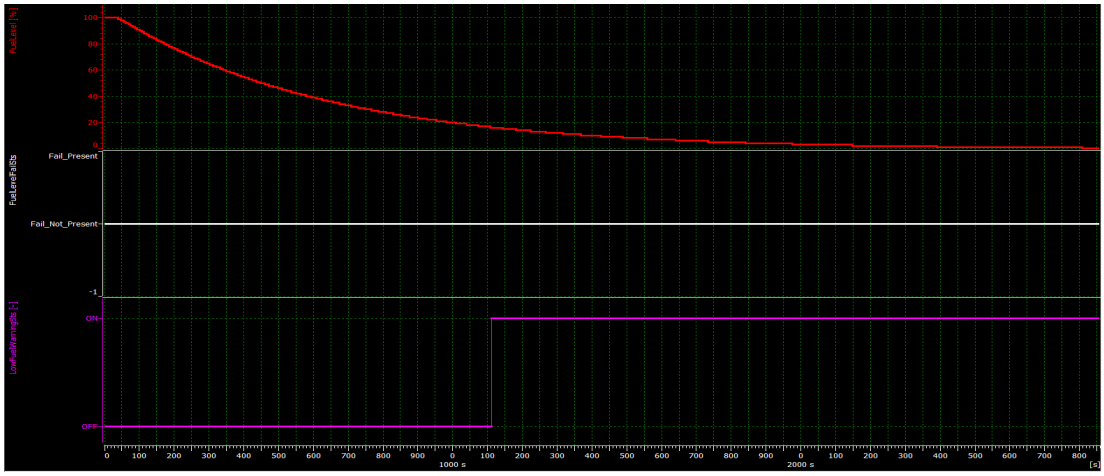


Figure 5.7: *Simulink* DYNAMIC test3.



(a) *DYNAMIC bench test3.*



(b) *Zoom of DYNAMIC bench test3.*

Figure 5.8: Bench Test 3.

5.3.2 Test Case 4

The only difference with the *Test Case 3* is related to the vehicle speed. In fact, in this case the vehicle runs at 127 km/h. Therefore, the filtering action is quicker than before and the fuel level indicator decreases more rapidly. In this situation, both the simulation end at about 1150 seconds.

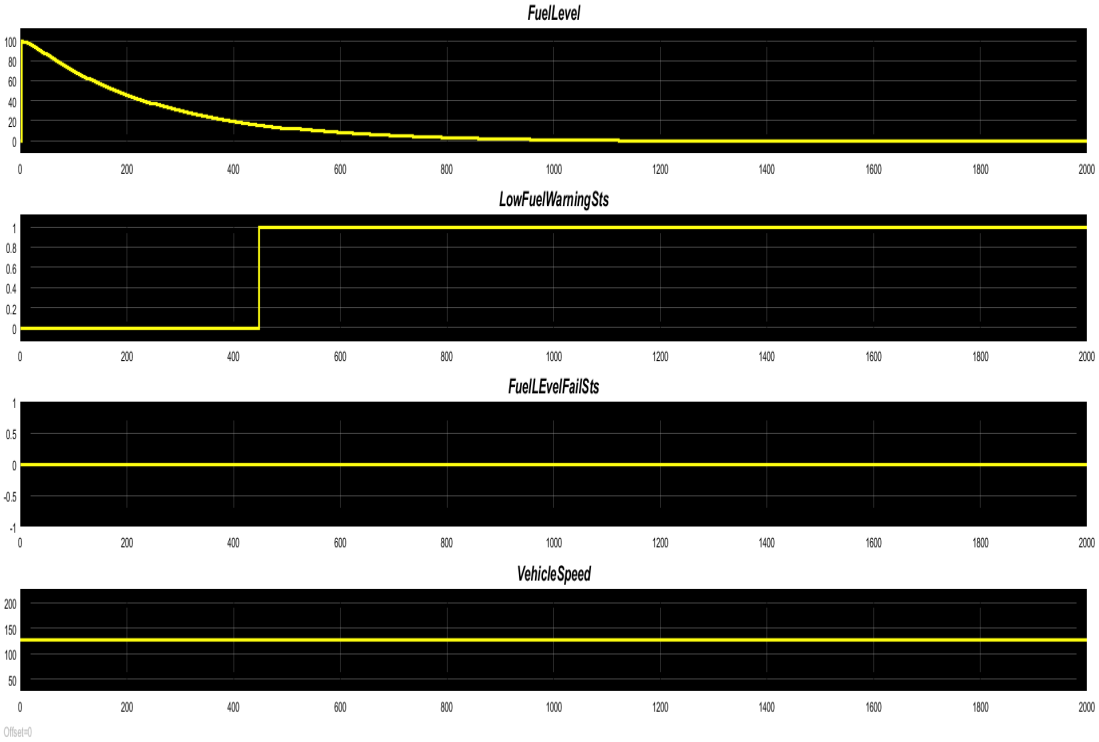
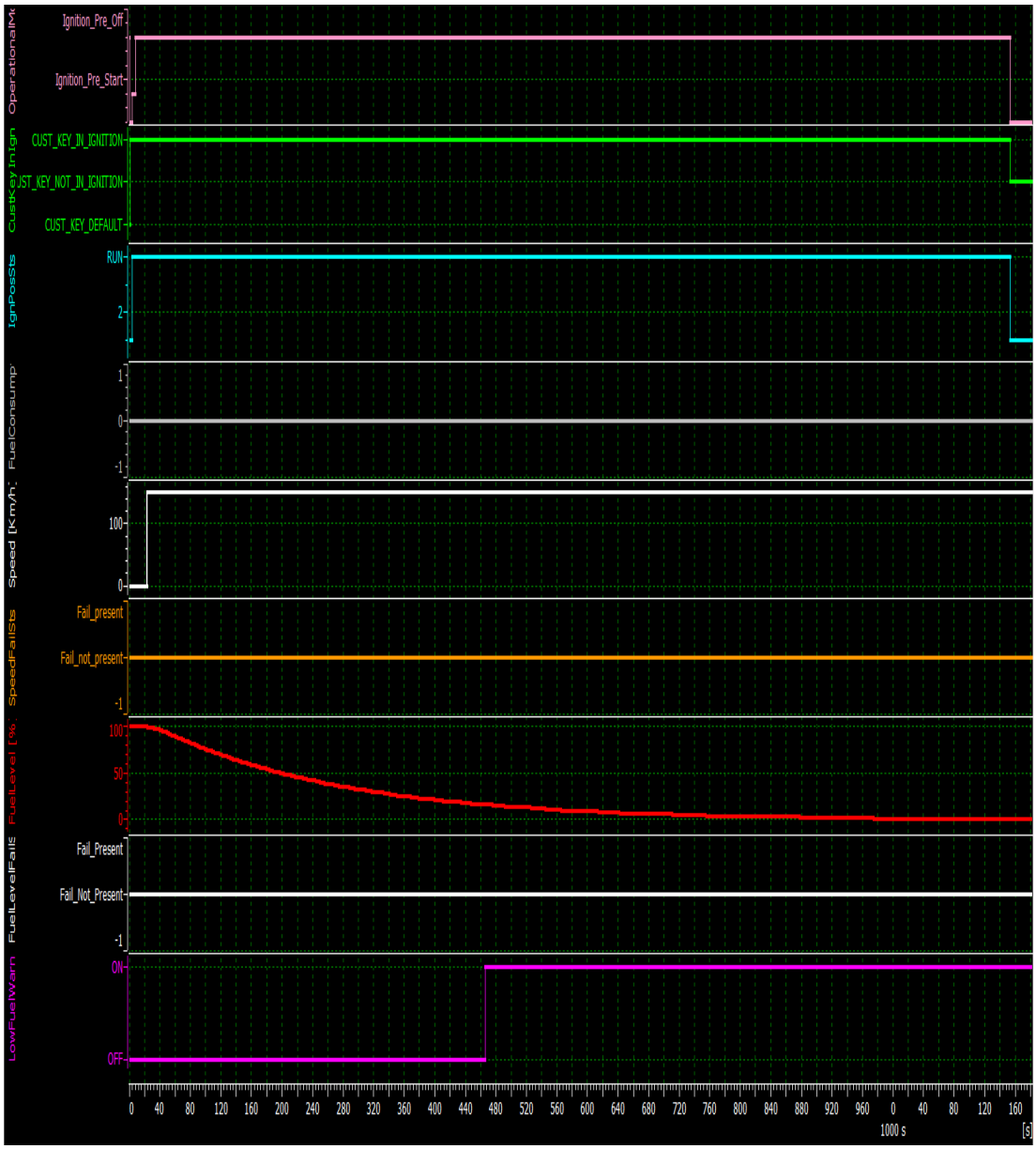
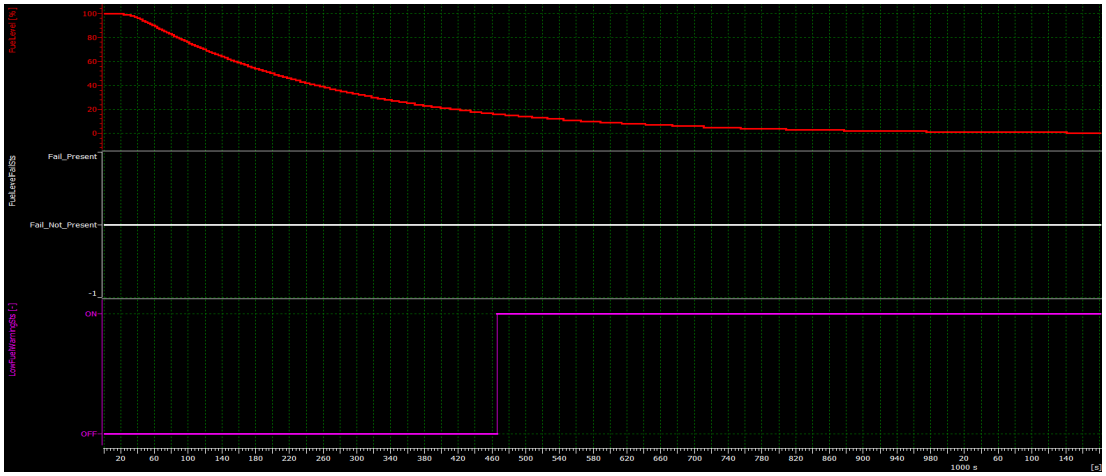


Figure 5.9: *Simulink* DYNAMIC test4.



(a) *RIFO bench test4.*



(b) *Zoom of RIFO bench test4.*

Figure 5.10: Bench Test 4.

5.4 COMPLETE

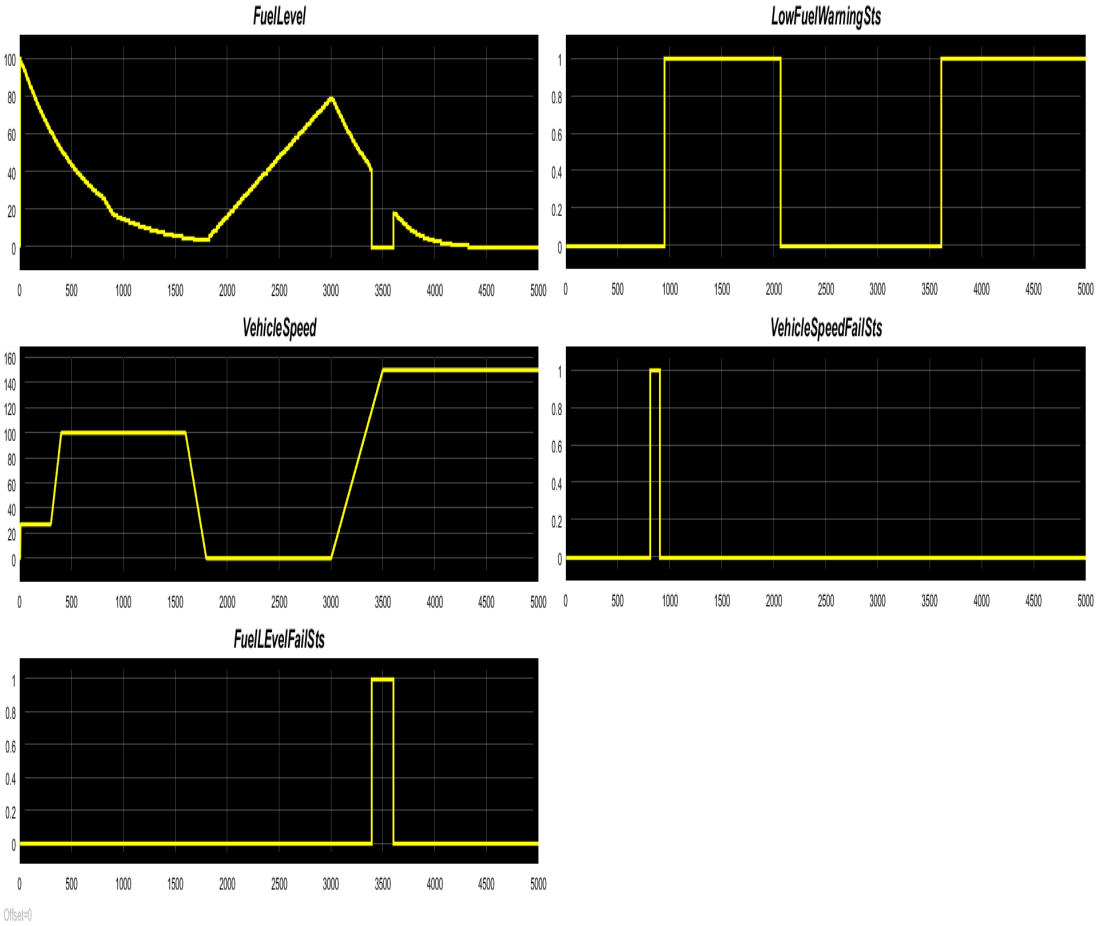
In comparison to the previous test cases, *Simulink* and bench test are going to be analyzed separately. Although the purpose of these tests is the same, they are representing the same scenario but different evolutions.

5.4.1 Test Case 5

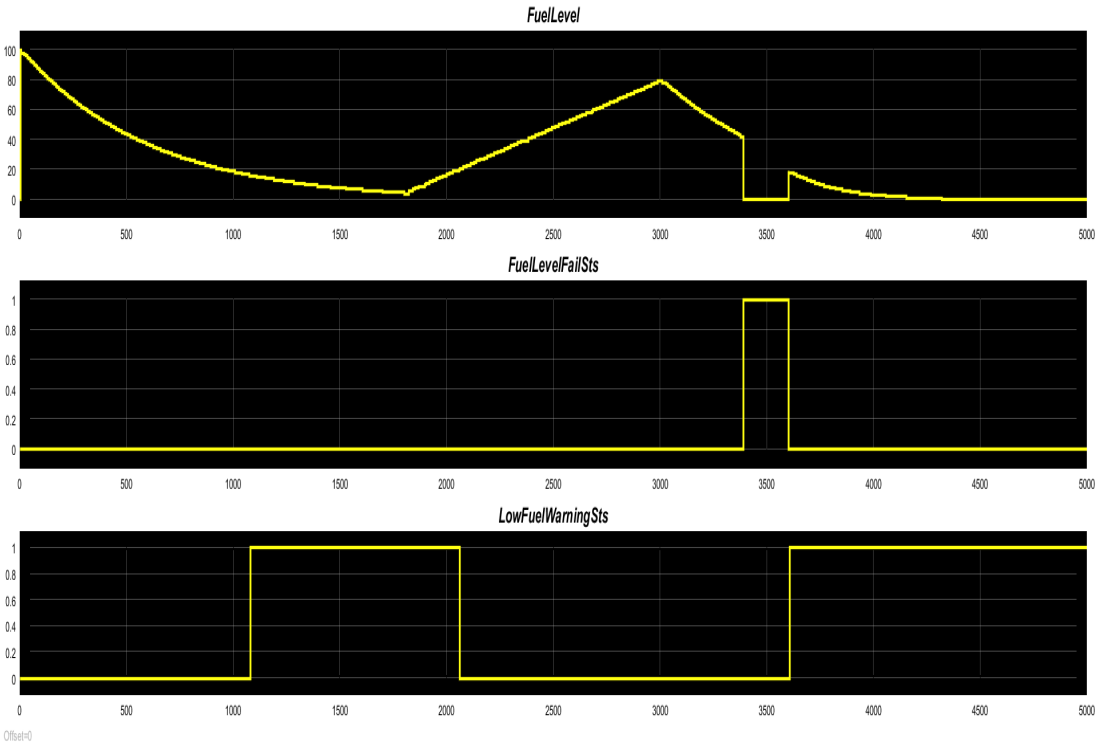
Simulink Test

The figure 5.11(a) describes a scenario where a vehicle is accelerating at 20 km/h and the fuel-tank is completely full. Due to the vehicle speed is lower than the *SpeedThreshold*, the fuel decreases according to the low filtering action but as soon as the speed increases and reaches 100 km/h the slope of the curve becomes steeper and the fuel falls more rapidly. Between 700 and 800 seconds, the significant change in slope is provoked by the validation of vehicle speed fail status that makes the filter act as in the low filtering mode. Once invalidated, the fuel keeps on decreasing with the high-filtering pace.

Regarding the second part of the figure, the refuelling of the vehicle makes the percentage rocket from about 4% to 100% within 1000 seconds. Henceforth, the car speed reaches 150 km/h and the logic outputs a reliable percentage until the level decreases to 0%. However, in order to simulate all possible input combinations, a fail on the sensor measurements is reproduced at 3500 seconds. In that condition, the fuel percentage plummets suddenly to 0% so that the driver notes the problem. In case of a temporary failure, once the issue is fixed, the dynamics continues its evolution.



(a) *COMPLETE Simulink test5.*

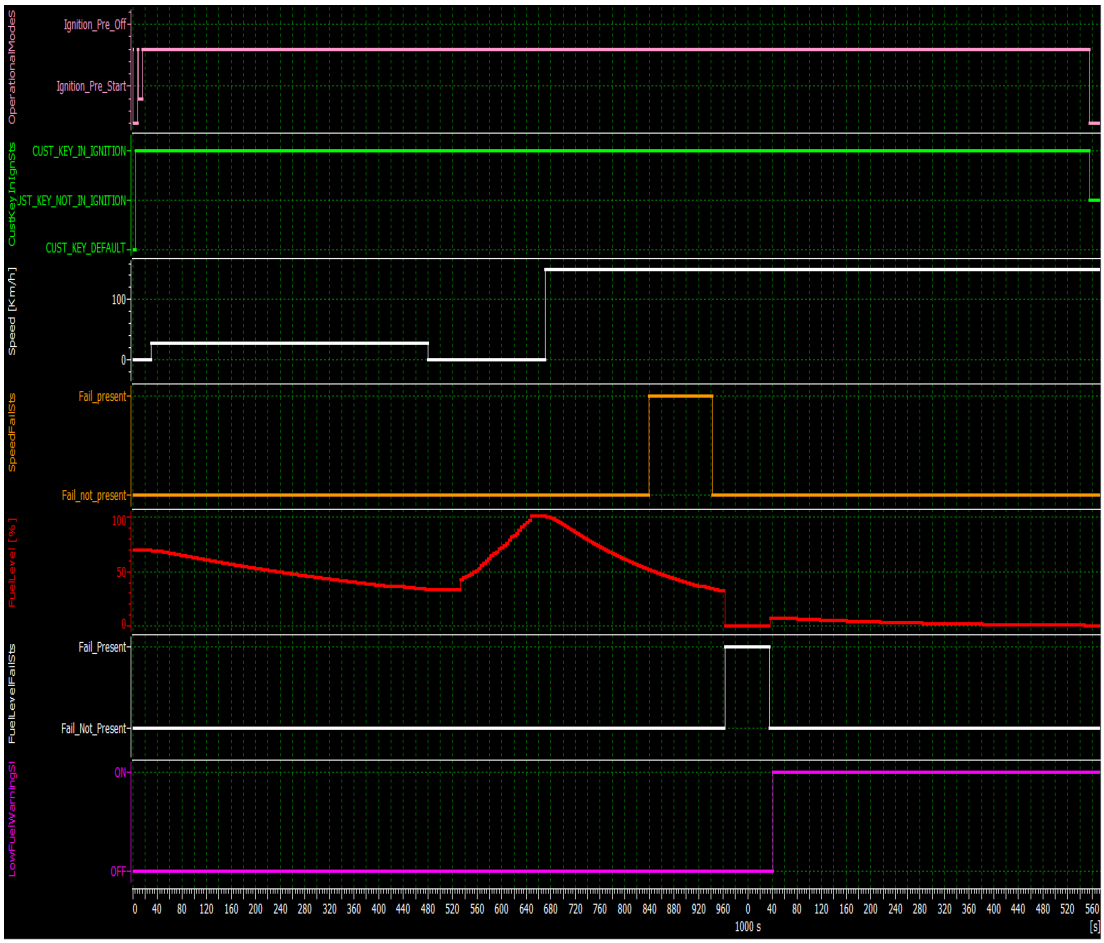


(b) *Zoom of COMPLETE Simulink test5.*

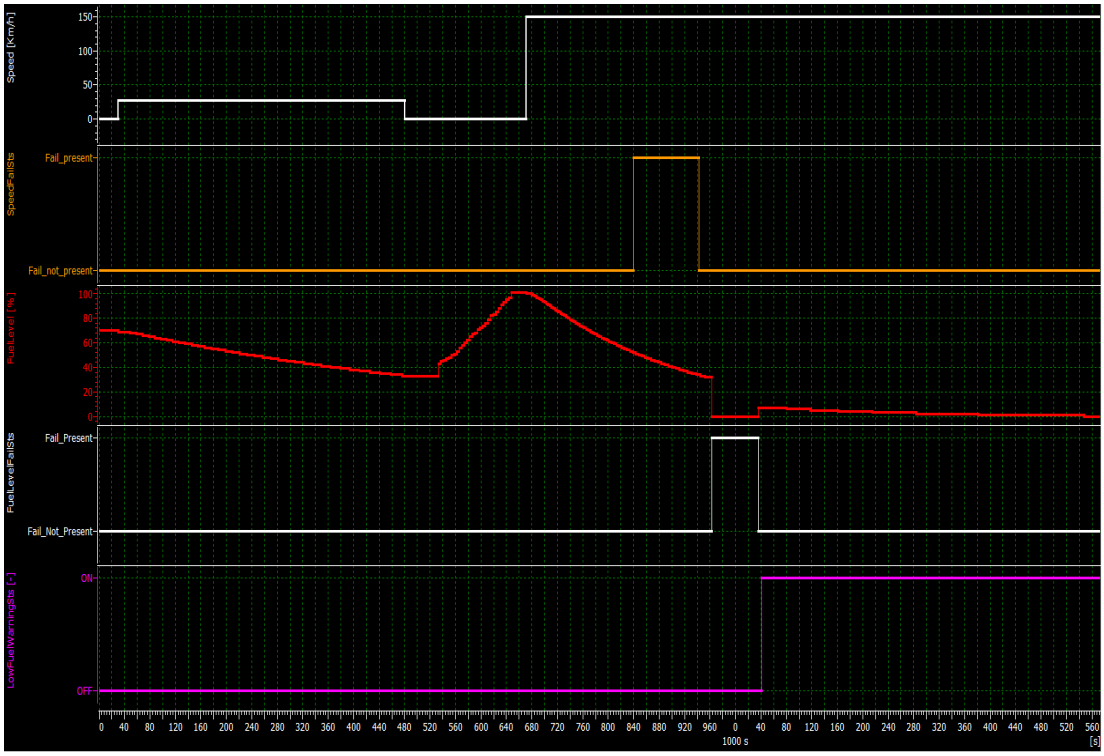
Figure 5.11: *Simulink* Test 5.

Bench Test

The bench test illustrates a similar situation to the previous *Simulink* case. The only difference is represented by the starting speed of the vehicle which is 70 km/h. The timeline events follow the same approach previously explained. In fact, once decreased to 40%, the vehicle stops and the refuel is performed until 100% is achieved. Afterward, the car accelerates at a constant speed of 150 km/h that makes the fuel level decrease according to the high filtering action. The presence of a failure in the *Sensor Management* makes the percentage sink to 0%. According to the graph 5.12(b), when the error is no longer validated, the logic works regularly as long as the car runs out all the fuel.



(a) *COMPLETE Bench test5.*



(b) *Zoom of COMPLETE Bench test5.*

Figure 5.12: Bench Test 5.

Conclusions

To sum up, the first chapter outlined how AUTOSAR changed the automotive world and the significant advantages related to this standard. For this reason, crucial pillars such as architecture, methodology, application interfaces and conformance test, were discussed and matched to future purposes of the consortium. Then, the second chapter dealt with requirements, sensors and transfer functions of the *Fuel Level Control* vehicle function and provided a high-level sketch of the model where the most important transitions and states were drawn.

The third chapter introduced the relationship between the *V-model* and both vehicle function requirements and objectives. In addition, the choice of the *Matlab/Simulink* environment for the model based design approach and the use of the *Stateflow* chart for the logic implementation are explained and used to realize the software model. Whereas, the remaining phases of the *V-model* were discussed in the fourth chapter where the working principle of software like *TargetLink*, *SOFTUNE Workbench* and *CANanalyzer* and hardware devices such as the CANcase and the hardware simulator were analyzed in depth.

What is noticeable from the final chapter is that the way, whereby the test had been worked out, follows a straight line of reasoning that goes from testing the single conditions of the vehicle function model to a complete test case including all the possible input variations. Although some of the test cases displays marginal and negligible differences related to bench instruments, the outcomes are in line with the expected results and attest all the study conducted throughout the thesis.

However, the development and the integration of a new functionality inside an automobile follows a different procedure in a company. In fact, people belonging to specific teams are involved in different stages of the vehicle function implementation. From an industrial prospective, it is typical that large-sized companies, simply labelled as *customer* in the figure 5.13, prefer ask suppliers for the development of an automotive functionality. Before explaining the stages behind this working relationship, it is important to outline that the customer could ask for the development of a functionality either from scratch or from an already-started implementation. Although the former enables the supplier to be free in the working organization, this manner is much more expensive for the customer. Conversely, providing a partial implementation is cheaper but provokes a greater effort to the supplier who needs to keep in touch with the client regularly to clarify any doubt. In fact, albeit the diagram illustrates just two one-side arrows toward the customer and supplier, during the development of a vehicle function

the contacts tend to be really frequent.

However, concerning the supplier internal organization, it mainly consists of three groups. The *Software Architecture Team* aims to describe and provide a well-detailed description of the vehicle function features and requirements. Afterwards, these are studied and implemented by the *Developing team*. However, in case of misunderstood or lack of information, both the groups will interact in order to tackle and fix any ambiguous description. While the *Testing Team* has the role in working out specific test cases to verify the correctness of the model carried out by the *Developing team*. Obviously, if there are unexpected results, the information will be sent back to the previous team. It is undeniable that before overcoming all the issues, the supplier chain is likely to be followed back and forth several times and if a doubt is not clarified by any team, the customer is advised in order to fulfill the error of the logic.

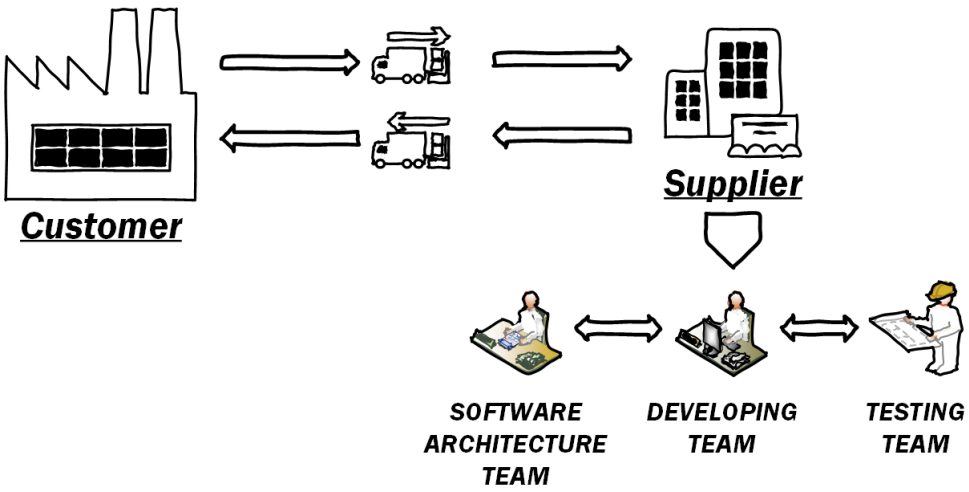


Figure 5.13: Client-Supplier relationship.

Acknowledgements

I would like to express my most sincere thanks to the following:

First and foremost, to my professor and supervisor Massimo Violante for helping me in finding a valid opportunity for the thesis and for his clever and efficient hints.

To the *TXT* team, especially Giuseppe, Lorena and Concetta, for the chance to work with a high-motivated, competent and skillful group of people over the whole internship period.

To my English teaching-staff Alison, Alan and Maura who did not only help me to improve language skills but also to purse my dreams in this years.

My deep and real thanks to all my friends for proving to me that distance is the key-factor in groups falling apart or strengthening the relationship. Today, our bond is as strong as ever.

To my sister and best friend Alessia whose attention to details together with her wise suggestions helped me to tackle and overcome critical issues.

To Manuela for being my travelling companion in our incredible journey and sharing the everyday routine with her smiles.

To my parents and role models, Patrizia and Giorgio, for teaching me to be curious and supporting me in any kind of choice especially within the last three years. Thank you for being always by my side regardless of the distance that divides us and showing me the real meaning of the words gratitude, love and respect.

List of Figures

List of Tables

Bibliography

[1] Stefan Voget, Michael Golm, Bernard Sanchez, and Friedhelm Stappert, "Application AUTOSAR standard", in *Automotive Embedded Systems Handbook* (Nicolas Navet and Francoise Simonot-Lion). CRC Press, Taylor & Francis Group, 2008, 2.1-2.25.

[2] AUTOSAR Web Team,
http:https://www.autosar.org/fileadmin/HOW_TO_JOIN/AUTOSAR_Introduction.pdf

[3] AUTOSAR,
https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_SWS_VFB.pdf

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

[5] Massimo Violante, Politecnico di Torino, "Operating System Architecture".

[6] Nico Naumann,
https://hpi.de/fileadmin/user_upload/fachgebiete/giese/Ausarbeitungen_AUTOSAR0809/NicoNaumann_RTE_VFB.pdf

[7] Simon Fürst,
http://st.inf.tu-dresden.de/files/teaching/ws08/ase/03_AUTOSAR_Tutorial.pdf

[8] David Haworth,
https://pdfs.semanticscholar.org/8095/f54a80f4f1990cf8b7006f82bc2776e78a85.pdf

[9] AUTOSAR,
https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_LayeredSoftwareArchitecture.pdf

[10] AUTOSAR,
https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-10/AUTOSAR_EXP_ParallelProcessingGuidelines.pdf

[11] AUTOSAR,
https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_PlatformDesign.pdf

[12] AUTOSAR,
`https://www.autosar.org/fileadmin/user_upload/standards/tests/1-2/AUTOSAR_EXP_AcceptanceTestsOverview.pdf`

[13] AUTOSAR,
`https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_EXP_AIUserGuide.pdf`

[14] Massimo Violante, Politecnico di Torino, "Introduction to AUTOSAR".

[15] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, "Operating-System Structures" in *Operating System Concepts*. John Wiley & Sons, 2004, 39-72.

[16] Simon Fürst, 8th Vector Congress, "The AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles".

[17] Dennis Kengo Oka, Phu H. Phung and Ulf E. Larson. *Vehicle ECU classification based on safety-security characteristics*. Road Transport Information and Control 2008 and ITS United Kingdom Members' Conference, DOI: 10.1049/ic.2008.0810, 2008.

[18] Real-Time System Laboratory,
`https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf`

[19] Bosch Professional Automotive Information, "Automotive networking" in *Bosch Automotive Electrics and Automotive Electronics. Systems and Components, Networking and Hybrid Drive*. Robert Bosch GmbH,, 2004, 82-87.

[20] Bosch Professional Automotive Information, "Bus systems" in *Bosch Automotive Electrics and Automotive Electronics. Systems and Components, Networking and Hybrid Drive*. Robert Bosch GmbH, 2004, 92-144.

[21] Bosch Professional Automotive Information, "Sensor measuring principles" in *Bosch Automotive Electrics and Automotive Electronics. Systems and Components, Networking and Hybrid Drive*. Robert Bosch GmbH, 2004, 232-308.

[22] Andreas Forsberg and Johan Hedberg.
`https://pdfs.semanticscholar.org/546c/e8a60a5db2054c7adb08a615d0ab060d88df.pdf`.

[23] Dr. Kevin Forsberg and Mr. Harold Moo,
`http://damiantgordon.com/Methodologies/Papers/System%20Engineering%20for%20Faster%20Cheaper%20Better.pdf`

[24] Michael Burke,
`https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/campaigns/portals/files/general-electric/stateflow-best-practices.pdf`

[25] Massimo Violante, Politecnico di Torino, "Introduction to model-based software design".

[26] Joonwoo Son, Ivan Wilson, Wootaik Lee and Suk Lee. *Model Based Embedded System Development for In-Vehicle Network Systems*. SAE Technical Papers, DOI: 10.4271/2006-01-0862, 2006.

[27] Mathworks. *Stateflow User's guide*. Mathworks, 2004.

[28] Arno Bergmann. *Benefits and Drawbacks of Model-based Design*. DOI: 10.14416, 2014.

[29] Massimo Violante, Politecnico di Torino, "Model Based Software Design".

[30] Conny Johansson and Christian Bucanac.
http://www.bucanac.com/documents/The_V-Model.pdf

[31] Tony Lennon.
http://www.ee.co.za/wp-content/uploads/legacy/AutT_Mod.pdf.

[32] TargetLink.
https://www.dspace.com/shared/data/bkm/targetlink_en/files/assets/common/downloads/TargetLink%20ProductiInformation.pdf.

[33] TargetLink.
https://www.dspace.com/shared/data/pdf/2018/dSPACE_TargetLink_Product-information_02-2018_E.pdf.

[34] Vector.
https://assets.vector.com/cms/content/products/canalyzer/canalyzer/Docs/Product%20Informations/CANalyzer_ProductInformation_EN.pdf.

[35] Vector.
https://assets.vector.com/cms/content/products/canalyzer/canalyzer/Docs/Product%20Informations/CANalyzer_ProductInformation_EN.pdf.

[36] Vector.
https://assets.vector.com/cms/content/products/VN16xx/docs/VN1600_Interface_Family_Manual_EN.pdf.

[37] SOFTUNE Workbench.
http://www.cypress.com/documentation/software-and-drivers/softune-ide-integrated-development-environment.

[38] SOFTUNE Workbench.
https://www.fujitsu.com/downloads/MICRO/fma/pdfmcu/CM71-00328-3E.pdf.