POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS
MASTER'S DEGREE COURSE IN MECHATRONIC ENGINEERING

MASTER'S THESIS
IN
MODEL-BASED SOFTWARE DESIGN

DEVELOPMENT OF THE CONTROL ALGORITHM FOR A RETROFIT RECONVERSION KIT

Supervisor:
Prof. Alessandro RIZZO

Company supervisor:
Luca BUSSI

Candidate:
Giorgio FANIZZI

ACCADEMIC YEAR 2023/2024

Abstract

The automotive industry is confronted with a multifaceted challenge: contending with intense competition, adapting to evolving standards, and managing increasingly complex components, all while addressing sustainability concerns. Global pollution and climate change demand solutions and electrification through Battery Electric Vehicles (BEVs) offers a promising path to carbon neutrality. Within this context, the innovative EVERGRIN project developed by “brain Technologies S.r.l.” plays a pivotal role. The general project proposal of the company is to develop an electronic device, a Vehicle Management Unit (VMU), that allows the rapid configuration of a new electric vehicle but can also be adapted to an existing vehicle, as part of a retrofit activity. This thesis is part of this ambitious company project and is divided into three parts: first, an overview of the EVERGRIN project is given with details and insights, followed by an examination of the relevant theories related to automotive standards, V-cycle, Model-In-the-Loop (MIL) phase and software testing; then, the operational part carried out for this thesis is revealed and illustrated step by step, starting from the writing of the system requirements in accordance with the current regulations in the industry (e.g. ISO/IEC/IEEE 29148:2018); it follows the definition of the general state machine describing the macro conditions assumed by a BEV during its different operational phases is discussed, with a focus on the development process and iterations; finally, everything is translated into software using Model-Based Software Design (MBDS) techniques, using Simulink environment for the creation of an interactive and real-time simulation, adjustable during execution from a customised dashboard. After a thorough analysis and discussion of the data, a final and overall consideration will be provided, tailored to future applications.

Index

Introduction ***4***

Chapter 1 – EVERGIN PROJECT ***7***

Chapter 2 – ISO 26262 .... ***17***

Chapter 3 – MODEL-BASED DESIGN ***30***

Chapter 4 – REQUIREMENTS AND STATE MACHINE ***40***

Chapter 5 – SOFTWARE IMPLEMENTATION ***56***

Chapter 6 – SIMULATION AND TESTING ***67***

Conclusion ***102***

Appendix ***104***

Bibliography ***120***

Acknowledgements ***121***

Introduction

Nowadays the automotive industry is navigating a complex and highly competitive terrain, marked by an abundance of players vying for market share. This challenge is further compounded by the continuous updates required to align with the ever-evolving rules and regulations imposed by various countries. At the same time, there is an urgent need to streamline production costs, demanding the optimization of production processes and the shrinkage of the whole time of production, a.k.a. time to market. Additionally, it is relevant to take into account the development of the necessary software that requires an enormous effort due to its increasing complexity over the years. Navigating this intricate and challenging landscape has become even more crucial due to the pressing need for sustainable solutions. As a matter of fact, one of the big problems of modern society is global pollution and climate change, so there is a demand to reach as soon as possible carbon neutrality (for instance, Europe is striving to become the world's first climate-neutral continent by 2050); investing in renewable electricity combined with Battery Electric Vehicles (BEV) would enable carbon neutrality also for transport. Life cycle assessments including the production of batteries raise questions regarding resource depletion including rare earth metals. Nevertheless, on average, BEVs are in most cases already on par with, or better than, conventional vehicles and with recycling and opportunities for resource substitution in batteries, resource scarcity is a tractable challenge [1]. The transition to BEVs is well under way in the automotive industry, with experts forecasting an increase in production over the coming decades. This shift though will not be smooth and orderly but, more accurately, it’s going to be lumpy and erratic: customer preference, pricing, regulations, and other market factors play a major role in

this changeover. So, there is no doubt that the transition to purely electric vehicles represents one of the most likely technological scenarios in the coming years, however, its full implementation faces some obstacles that have not been completely overcome yet:

- the cost of a fully electric vehicle, including batteries, remains around 10 to 20,000 euros more than a conventional car of the same segment;
- the supply chain of components and subsystems providers is not ready yet to offer optimized solutions according to automotive standards;
- there are numerous vehicles with Euro 3 or Euro 4 emission standards that, despite being perfectly functional, can no longer be used in lots of urban centres (depending on the country's regulations); their disposal, therefore, will represent a cost.

Unless these issues are addressed, at least to some extent, the adoption of electric vehicles will struggle to take off. This is where the groundbreaking project, called EVERGRIN, advanced by "brain Technologies S.r.l." comes in. The general project proposal of the company is to develop an electronic device, a Vehicle Management Unit (VMU), that permits rapid configuration of a new electric vehicle, but can also be adapted to an existing vehicle, as part of a retrofit activity, i.e. the conversion of a polluting conventional vehicle into a pure electric one allowing a considerable cut in the total cost.

This thesis is part of this ambitious company project, and it will be developed according to the following steps:

- initially, an overview of the EVERGRIN project with details and insights will be provided, followed by an examination of relevant theories related to the automotive standards, V-cycle, development phase and In-the-Loop testing;
- then the operational part that was carried out for this thesis work will be exposed and illustrated step by step; starting from the writing of the system

requirements, in accordance with the current regulations in the industry (e.g. ISO/IEC/IEEE 29148:2018); afterwards, the definition of the general state machine describing the most significant macro conditions assumed by a BEV during its various operating phases, with a focus on the development process and iterations, will be discussed;

- lastly, everything will be translated into software using Model-Based Software Design (MBSD) techniques within Simulink resulting in an interactive and real-time simulation adjustable during the execution from a custom-made dashboard. Following a thorough analysis and discussion of the data, a final and comprehensive consideration tailored towards future applications will be provided.

Chapter 1

EVERGRIN PROJECT

A full evaluation of the significance of this thesis and its contribution to the field of electric vehicles requires a close examination of the EVERGRIN project's goals and how it addresses the challenges of electric vehicle adoption. As mentioned before, at the core of this project lies the development of the Vehicle Management Unit (VMU), a smart device that simplifies the setup and customization of electric vehicles. This technology applies to both newly built models and existing gasoline cars undergoing conversion through a retrofit process. The VMU, developed according to the latest automotive standards, will consolidate functionalities currently found in various subsystems like the Battery Management System (BMS) and inverter. This centralization not only enables the development of ISO 26262 compliant safety features but also reduces the number of required minimal subsystems, leading to a significant cost saving. Positioned as a crucial component for electric vehicle architecture, the VMU targets both new vehicles (M1, N1, offroad, and L quadricycles) and the retrofit market. In the latter, it empowers specialized workshops to convert polluting vehicles to electric at a fraction of the cost of new electric vehicles, offering adequate performance for urban use. The project concludes with the construction of a retrofitted electric vehicle that acts as a mobile laboratory. This real-world testing environment allows us to measure key metrics like range, performance, and cost savings, effectively demonstrating the VMU's success in achieving its scope.

To be able to understand the functionality of the system, it is important to first define the logical architecture, which includes identifying the key components and their interactions. The project utilizes a Modular Technical Model (MTM) as a pillar throughout various phases. This digital representation of the vehicle serves as a "virtual prototype" for simulating overall dynamics in MATLAB/Simulink and for defining the control logic using a model-based control design approach. The MTM's versatility extends to automatic code generation, facilitating both rapid prototyping and the production control unit within the Electric Retrofit Kit (ERK) system. Furthermore, the model guides the preliminary sizing of key components like the motor, inverter, and battery pack. It additionally integrates with the EVERGRIN project's test bench in both kinematic and dynamic modes. This seamless integration between the virtual prototype and test bench enables real-time validation and development of the ERK system's control logic directly on the VMU control unit. The overall development process, encompassing both the ERK system and specifically the VMU control logic, follows an extended V-cycle (more on that later) incorporating the rapid prototype phase on the test bench. This iterative approach ensures an optimized final realization of the actual vehicle. A detailed representation of this process is provided in the accompanying figure:

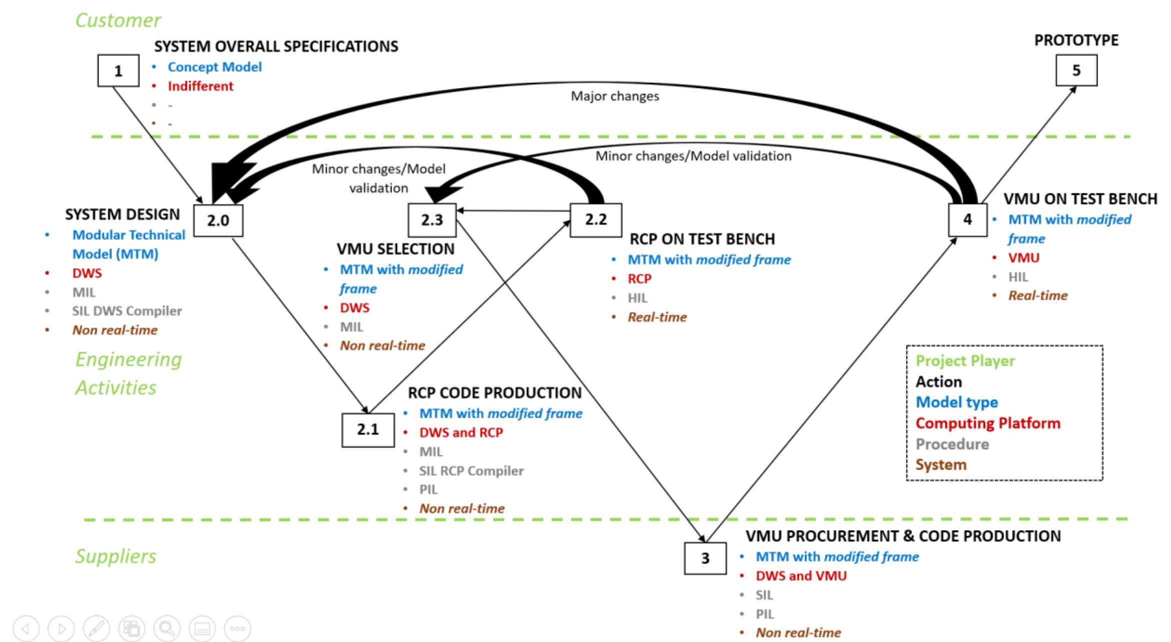


Figure 1 - Extended V-cycle of the development process

The extended V-Cycle employs a rigorous multi-stage verification process for the developed control logic. This process goes through distinct execution modes, which are listed below:

- Model-In-the-Loop (MIL) phase: the control logic is initially developed and verified through a dynamic model using various simulations on a high-performance computer, the Development WorkStation (DWS);
- Software-In-the-Loop (SIL) & Processor-In-the-Loop (PIL) phases: the generated code previously compiled with the DWS is subsequently ported to a system for Rapid Control Prototyping (RCP from dSpace), which is a processor that will run and test the code in real time but in a simulated environment;
- Hardware-In-the-Loop (HIL) phase: finally, the code is compiled for the target processor and uploaded to the VMU. Functional testing of the VMU occurs on the test bench, where it interacts with real physical components of the ERK system for comprehensive validation.

The following figure depicts the connections and interactions between the various modules of the whole vehicle's dynamic model. It also includes the user interfaces and the user himself. Additionally, the figure highlights a set of monitors that can be used to verify and track the behaviour of the individual modules. These monitors may be partially or fully available in the physically realized system.

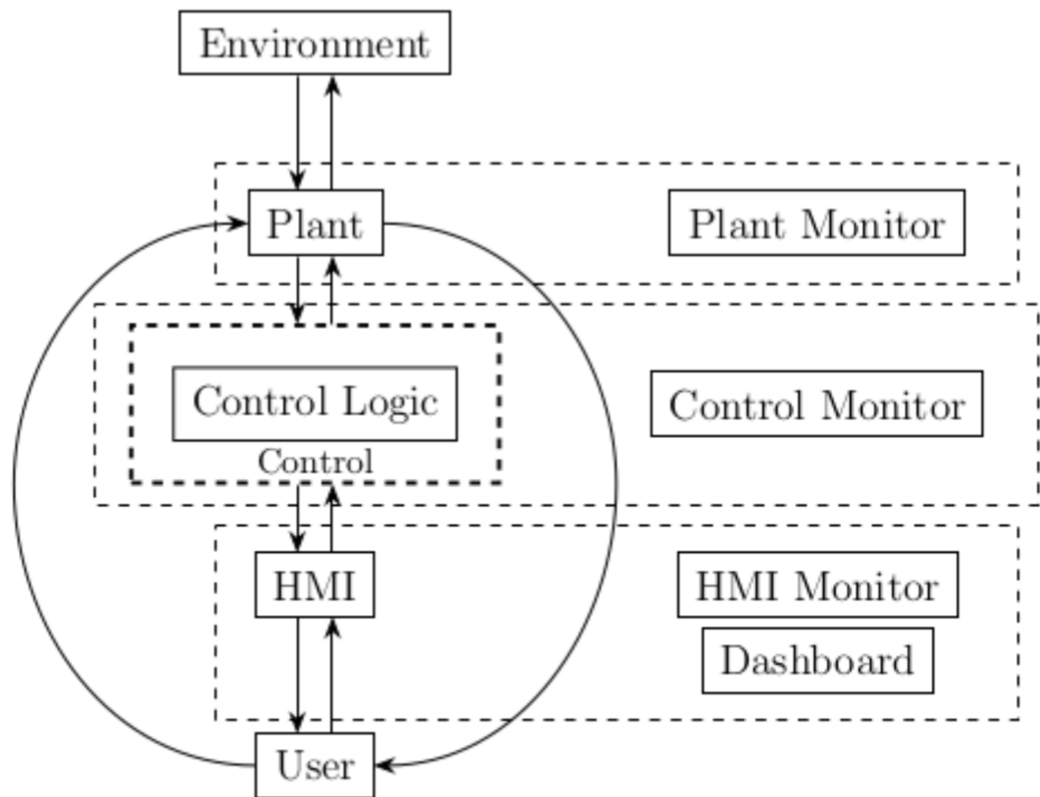


Figure 2 - Connections and interactions between the modules of the vehicle's dynamic model

The MTM is built in MATLAB/Simulink using nested subsystems and reusable components (referenced blocks). This approach allows for efficient development and separate management of complex sub-models by specialists, like those focusing on vehicle dynamics or automatic controls, e.g. the motor model or the battery model. In order to fulfil its dual purpose effectively, the vehicle dynamics controller requires a

special structure: serving as both a simulation tool and the foundation for generating the VMU control unit code; so, the control module comprises two distinct sections: a frame encapsulating the input and output interfaces, and the core control logic section. The next figure shows the general diagram of the entire control module:

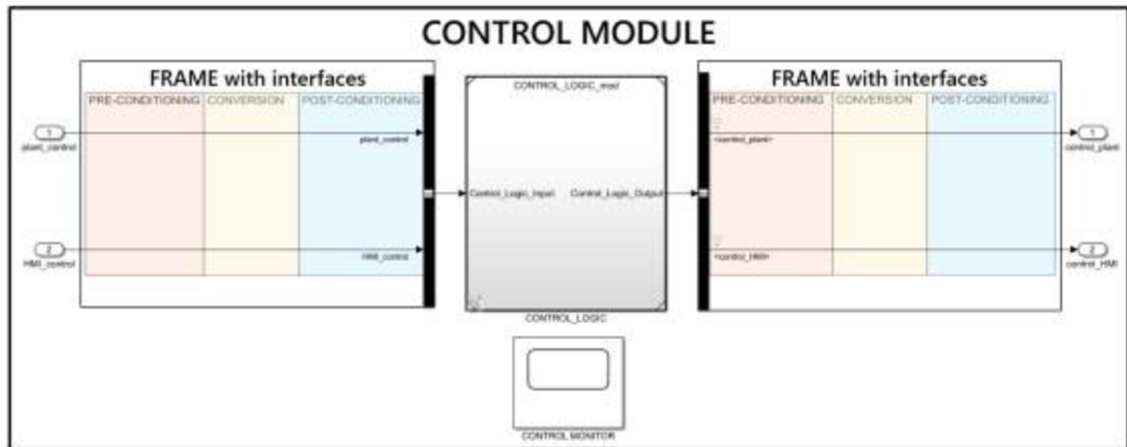


Figure 3 - General diagram of the control module

Once the whole logical architecture was defined, the system requirements specification phase was then conducted for the vehicle requalified with the ERK. The ISO 29148 System and Software Engineering requirements standard was used as a reference, with a specific focus on a "dynamic state machine" control logic setup. The first step of this approach involved identifying the possible states of the system and the transition conditions between them; an example proposal is shown in the following figure:

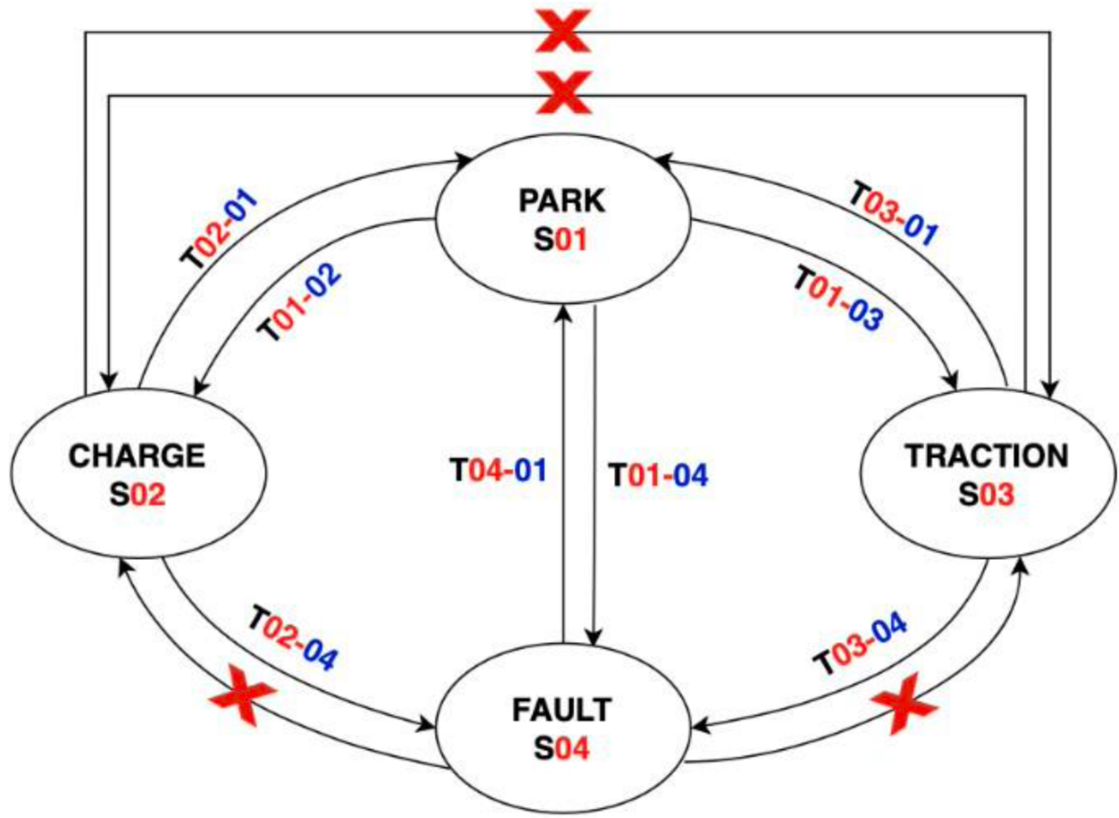


Figure 4 - Possible states of the system and the transition conditions between them

Following state and transition identification, the system's behaviour is rigorously defined through detailed functional requirements and transition conditions. The model-based approach demands a meticulously designed and efficient dynamic model for the entire system. As mentioned earlier, this is achieved through a modular dynamic model, the MTM, built in MATLAB/Simulink. This model leverages nested and referenced blocks (subsystems and referenced) for simplified separate management of various components, supporting diverse skill sets and use cases.

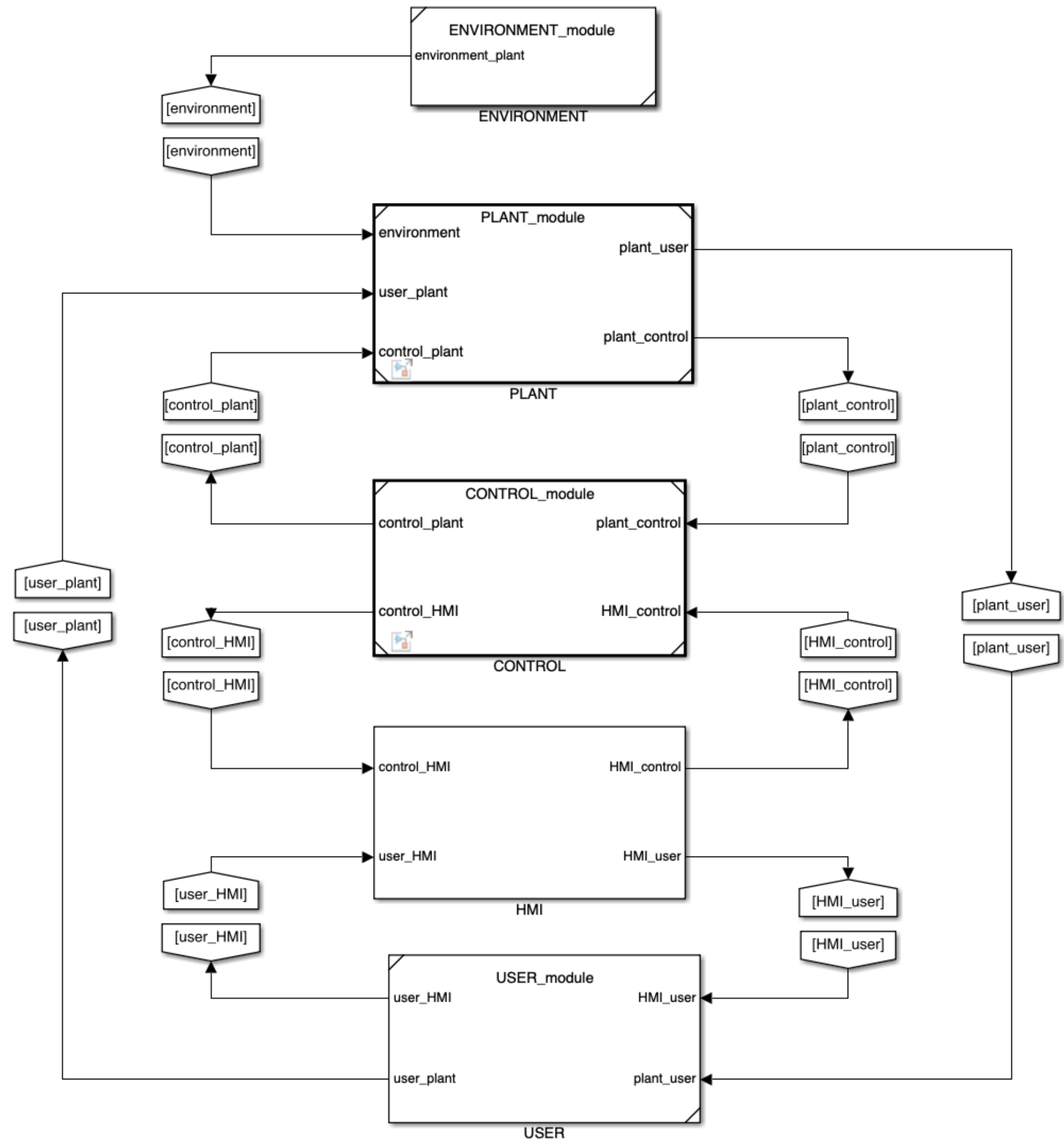


Figure 5 - Simulink scheme of the full modular technical model (MTM)

The vehicle model, referred to as the "plant" in this context, incorporates the following dynamics:

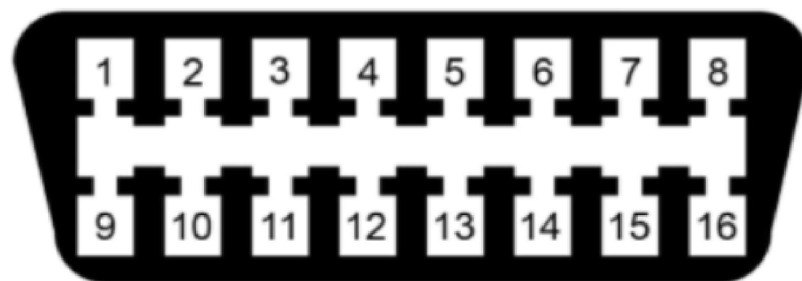
- longitudinal motion parameterized using the "coast-down" method;
- electrical and thermal characteristics of the battery;
- efficiency maps of the electric motor torque and drive system.

Therefore, the model is implemented as a "referenced block" to facilitate separate development and validation from the rest of the comprehensive modular technical model. The Human-Machine Interface (HMI) serves as the user's gateway to controlling the vehicle, comprising functions from parked state to dynamic driving scenarios. During the design phase, a virtual control panel facilitated code validation and provided valuable insights for shaping the final physical interface. To accurately simulate the vehicle's dynamic behaviour under real-world driving conditions, a driver model was incorporated to follow a predefined speed reference. Throughout the project, the Simulink/Stateflow environment played a crucial role in verifying adherence to all functional and timing specifications. This included, for instance, closely monitoring transitions between system states. The integration phase ensured seamless collaboration between the newly developed control system and the vehicle's pre-existing systems. This involved a series of targeted tasks, including:

- Controller Area Network (CAN) sniffing and integration, so monitoring and analysing the CAN network to ensure proper communication between the control system and other vehicle components;
- signal tracing, where useful signals were identified and tracked within the vehicle to provide the control system with critical data for decision-making;
- control model integration, which consists of the integration onto the VMU of the control model itself, enabling real-time execution and coordination of control algorithms.

The initial stage of integration involved identifying the CAN frames used by the Fiat Panda II. To achieve this, a Raspberry Pi 3 equipped with an MCP2515 driver served as the interface to the car's CAN bus. This cost-effective solution took advantage of the affordability of Raspberry Pi compared to other options. The reverse engineering process involved connecting the Raspberry Pi with the MCP2515 CAN interface to the car's OBD2 port. OBD stands for On-Board Diagnostic and is the standardized system that allows an external electronic device to interface with a car's computer system. In this case,

the OBD2 connector is located under the dashboard on the driver's side. OBD2 allows you to have complete control over engine parameters and monitor other parts of the car such as the chassis and accessories. The structure of the OBD2 connector is shown in the next diagram:



OBD2 Pin	Description
2	SAE J1850 Bus+
4	Chassis
5	Signal Ground
6	CAN High
7	ISO9141 K Line
10	SAE J1850 Bus-
14	CAN Low
15	ISO9141 L-Line
16	Vehicle Battery Positive

OBD-II connector pinout

Figure 6 - OBD2 connector with pins description

Following the completion of the configuration phase for the Raspberry Pi, a remote connection was established through VNC Viewer software to facilitate the subsequent CAN message sniffing process. Three essential software utilities dedicated to CAN communication were employed during the sniffing phase:

- *can_viewer* offers real-time visualization of incoming CAN messages;
- *can_logger* enables the recording of received messages to a designated file;
- *can_player* facilitates the replay of CAN frames stored within a log file.

To verify the operational status of the CAN interface on the Raspberry Pi, the car engine was activated, and the *can_viewer* command was executed to confirm the presence of CAN traffic. The baud rate was identified through the examination of standardized speeds, with implausible data transmissions being discarded; the determined speed was around 50 kbps. The *can_viewer* script underwent repeated executions, enabling the continuous monitoring of messages generated by various car operations, including the activation of lights, steering manipulation, and the engagement of high and low beam functionalities. To decipher the intent behind these messages and potentially engage in ethically responsible car manipulation, numerous message dumps were acquired through the utilization of the *can_logger* script. In conclusion, the *can_player* command was employed to replicate the captured messages, constituting the fundamental component of the CAN analysis. Following its development, the MTM model was integrated into the vehicle management unit using drivers and the Real-Time Operating System (RTOS). This integration was facilitated by the code generation capabilities provided by the MathWorks suite, effectively translating the MTM model into executable C code. The control software underwent rigorous testing through a two-step process aligned with the V-cycle extended for prototyping. This process involved Software-In-the-Loop (SIL) and Hardware-In-the-Loop (HIL) testing, both widely adopted in the automotive industry as essential phases of the design process. The rigorous methodology employed guaranteed a high level of safety and robustness for the algorithm in all usage scenarios. The V-cycles compliance with ISO 26262 facilitated the design of the VMU and internal software, ensuring an easier homologation process for the entire kit, fulfilling the purpose of the project [2].

Chapter 2

ISO 26262

Building upon the previous chapter, it is crucial to reiterate the project's reliance on model-based techniques throughout its entirety, from definition to development and final implementation. This approach, coupled with strict adherence to the latest international automotive standards, was essential for the project's success. ISO 26262 is the international standard for functional safety that governs Electrical and Electronic (E/E) systems in road vehicles (cars, motorcycles, heavy vehicles), implementing safety functions. It defines the requirements for the whole lifecycle of the system (ranging from the specification, to design, implementation, integration, verification, validation, and final production release), achieving a level of safety proportionate to the identified risks, ensuring hardware and software components meet the necessary standards. The adoption of ISO 26262 empowers Original Equipment Manufacturers (OEMs) and their suppliers with compelling advantages:

- enhanced safety assurance, so demonstrate and guarantee the overall safety of vehicles and related systems, upholding the stringent standards of the standard;
- competitive edge, gaining a significant advantage by accurately interpreting and implementing the requirements, propelling then your position in the marketplace;

- reduced risk, minimizing the potential for harm to individuals and mitigating product non-acceptance due to safety concerns;
- cost-effectiveness, preventing costly product recalls and potential reputational damage arising from inadequate safety measures;
- global market accessibility; streamline entry into international markets by adhering to the relevant international regulations established by the standard.

The International Organization for Standardization (ISO) collaborates closely with the International Electrotechnical Commission (IEC) in developing standards. Building upon IEC 61508, the generic functional safety standard for E/E systems, ISO released the initial version of ISO 26262 in 2011, specifically addressing functional safety in road vehicles. The standard was further revised in 2018 to reflect advancements in automotive technology and industry best practices. Originally published as a Draft International Standard (DIS) in June 2009, ISO 26262 has gained significant traction within the automotive industry. This early availability as a public draft is partly responsible for its widespread adoption, as legal professionals could consider it the "state of the art" in automotive functional safety.

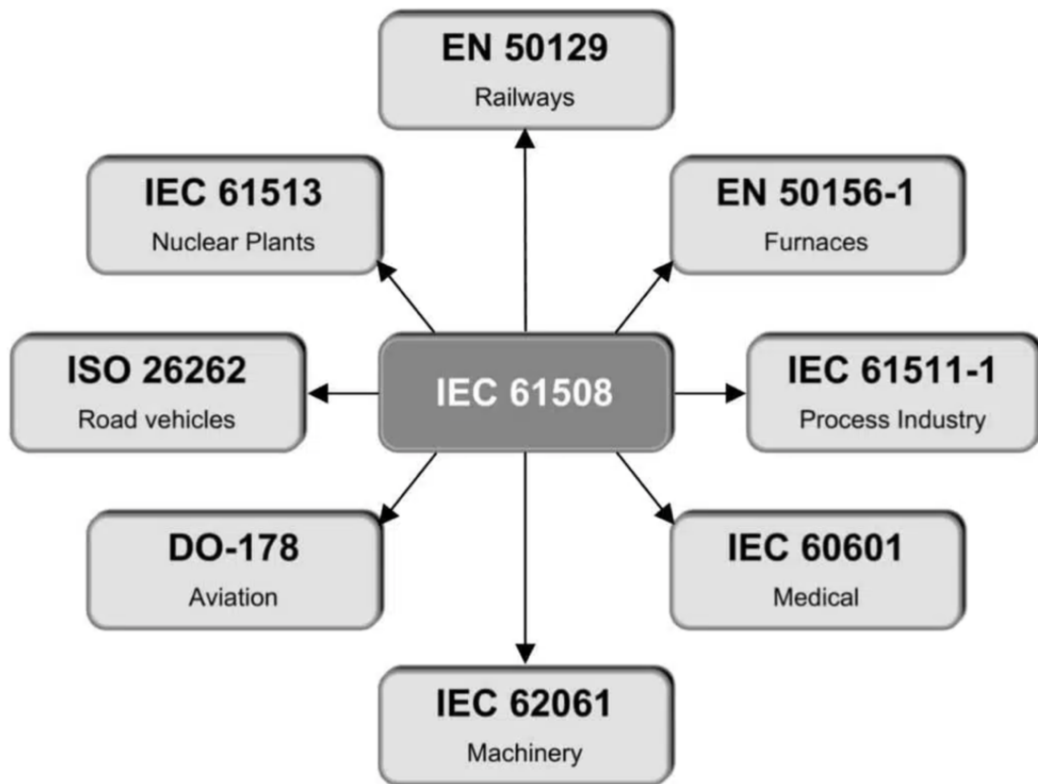


Figure 7 - Dependence of ISO 26262 from IEC 61508

Functional safety refers to the deliberate inclusion of safeguards in embedded systems to minimize the risk of harm to users. It's critical to mitigate potential hazards and emphasize risk-based implementation, where the level of safety measures scales with the potential consequences. While not directly enhancing core functionality, functional safety is a necessary investment in ensuring the well-being of end users. To achieve this, the standard provides an automotive-specific risk-based approach for determining risk classes, called Automotive Safety Integrity Levels (ASILs), from A (the lowest) to D (the highest), plus Quality Management (QM). Uses ASILs to specify the item's necessary safety requirements for achieving the freedom from unacceptable risk of physical injury or of damage to the health of people (harm) either directly or indirectly, through damage to property or to the environment [3].

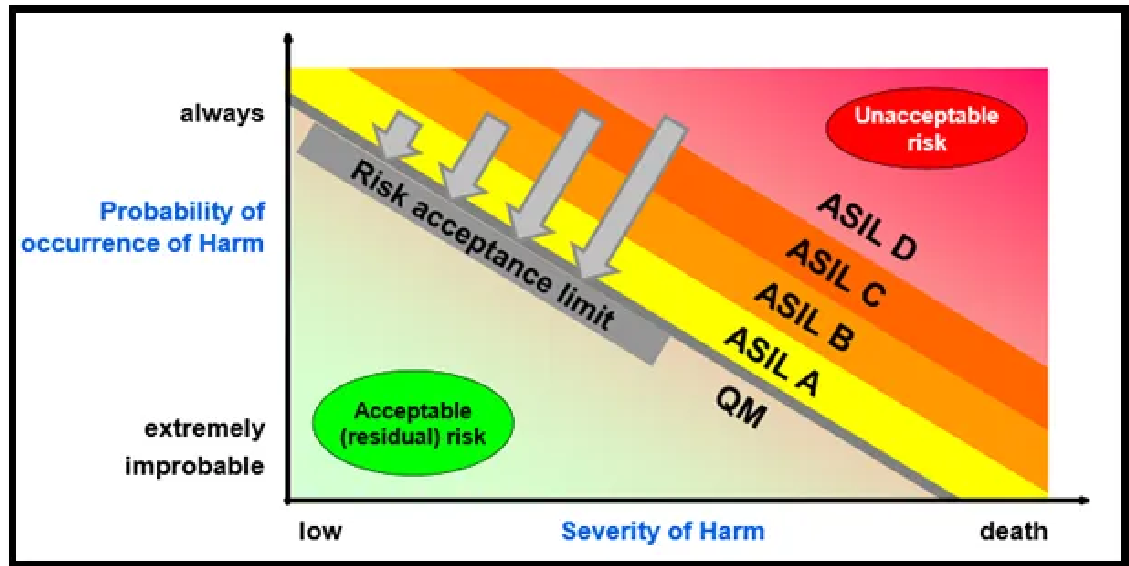


Figure 8 - Automotive Safety Integrity Levels (ASILs) scheme

Assigning an Automotive Safety Integrity Level (ASIL) to a potential source of harm (hazard) requires conducting a Hazard Analysis and Risk Assessment (HARA). This method identifies and categorizes potential hazards associated with an item (system or array of systems or a function to which ISO 26262 is applied), establishes safety goals (top-level safety requirement as a result of the HARA), and determines the necessary ASILs to prevent or mitigate these hazards and achieve an acceptable level of residual risk (combination of the probability of occurrence of harm and the severity of that harm).

It is now possible to direct our focus to the ISO 26262 safety life cycle, which serves as a structured framework for the development and maintenance of safety-critical systems within the automotive industry. This life cycle prioritizes continuous risk management, ensuring the identification, mitigation, and control of potential hazards throughout the entire development process.

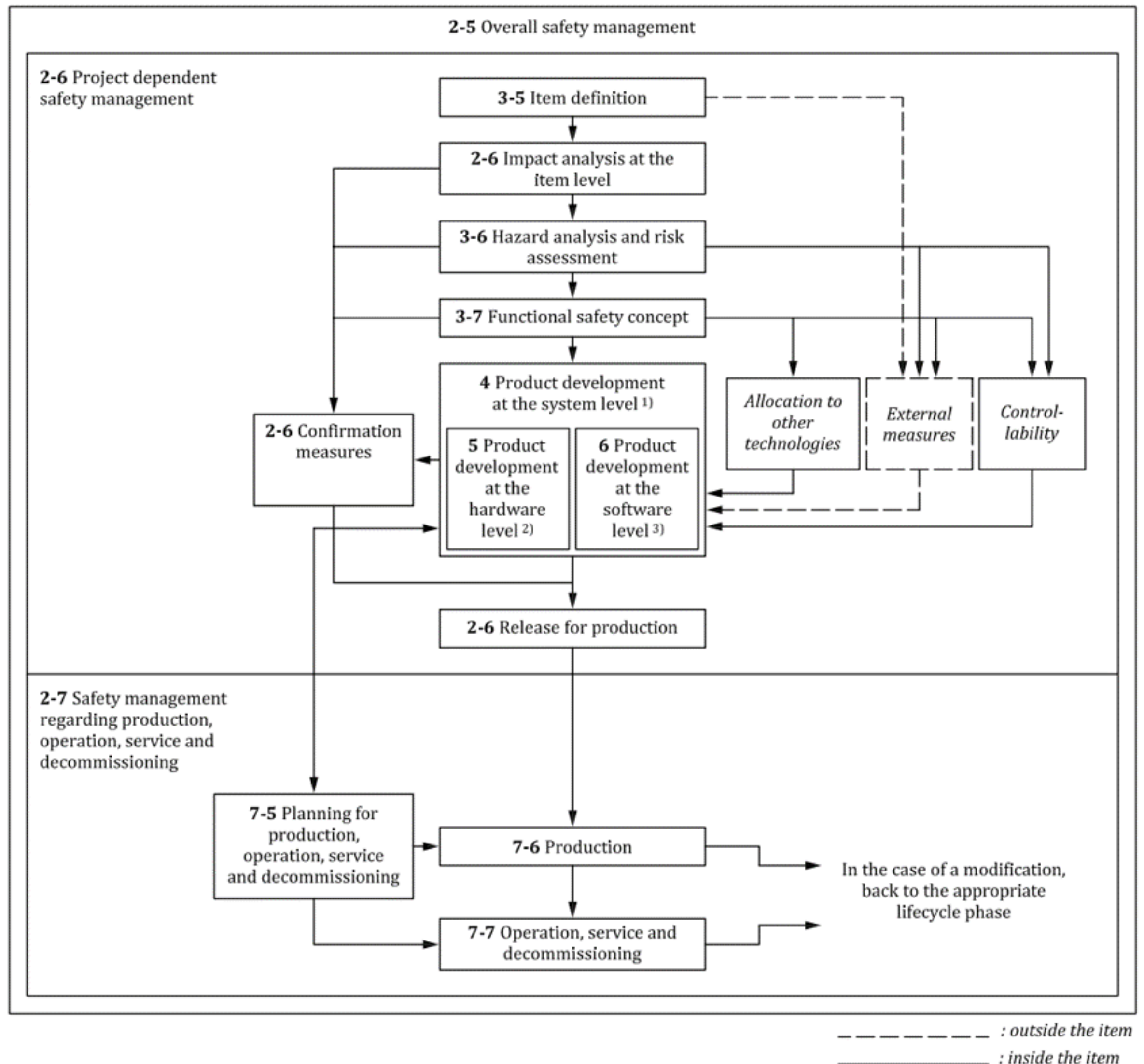


Figure 9 - ISO 26262 safety lifecycle

The initial stage of the standard's safety lifecycle, as shown in the figure above, involves defining the Item Under Test (IUT); this includes its purpose, functionalities, and interactions with the surrounding environment and other components. This information lays the groundwork for the subsequent HARA, which consists of four key parts:

- situational analysis: describe operational situations and operating modes in which malfunction results in hazardous events;
- hazard identification and classification: determine hazardous events, using appropriate techniques like brainstorming, checklists, quality history, Failure Mode and Effect Analysis (FMEA), for relevant combinations of operational situations and determine their consequences;
- ASIL determination: classify each hazardous event based on specific parameters;
- safety goals determination: definition of a safety goal for each hazardous event with ASIL, so not for QM.

Focusing on the third part, for each identified hazard, the ASIL classification is based on the three main parameters:

- *Severity* - estimates the extent of harm that can occur to one or more individuals in a potentially hazardous event. The severity is rated from S1 (light injuries) to S3 (fatal injuries);
- *Exposure* - assesses the likelihood of being in a driving situation that can be hazardous when it coincides with the identified hazard or failure mode. The exposure is rated from E1 (very low probability) to E4 (high probability);
- *Controllability* - evaluates the ability to avoid specific harm through the timely reaction of the involved person. The controllability can be rated C1 (simply controllable) to C3 (difficult to control).

For a more detailed look at each of the parameters, including their exact levels and descriptions, please refer to the following tables:

Severity level	*Title*	*Description*
S1	Light	Light and moderate injuries to the driver or passengers or people around the vehicle
S2	Severe	Severe and life-threatening injuries to the driver or passenger or people around the vehicle or in other surrounding vehicles
S3	Fatal	Life-threatening and fatal injuries to the driver or passenger or people around the vehicle or in other surrounding vehicles

Table 1 – Severity levels

Controllability level	*Title*	*Description*
C1	Simply controllable	99% or more of all drivers or other traffic participants are usually able to avoid harm
C2	Normally controllable	90% or more of all drivers or other traffic participants are usually able to avoid harm
C3	Difficult to control	Less than 90% of all drivers or other traffic participants are usually able, or barely able, to avoid harm

Table 2 – Controllability levels

Exposure level	*Title*	*Description*
E1	Very low probability	Less than 0.1 % of operating time
E2	Low probability	From 0.1% to 1% of operating time
E3	Medium probability	From 1% to 10% of operating time
E4	High probability	10% or more of operating time

Table 3 – Exposure levels

The combination of the three parameters will lead to determination the of the ASIL level for every hazardous event.

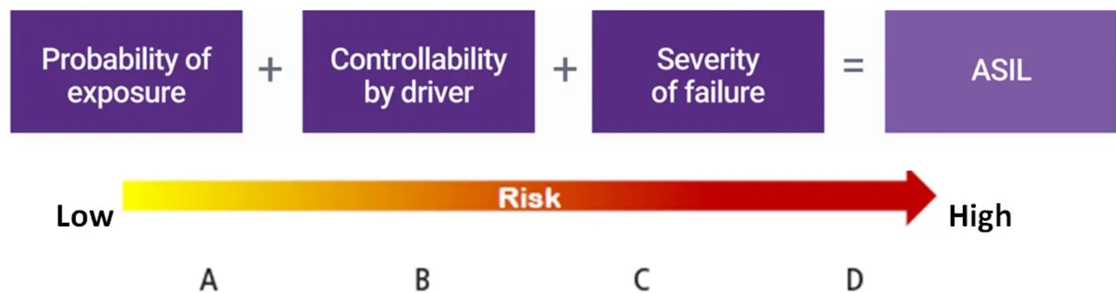


Figure 10 - ASIL determination

The interplay of severity, exposure, and controllability leads to the QM rating; in this case, the system is not safety relevant. However, for safety-relevant hazards (rated ASIL A, B, C, or D), a specific safety goal, expressed in functional terms that specify the required actions to mitigate the hazard, must be defined.

Severity	Exposure	Controllability		
		C1 (Simple)	C2 (Normal)	C3 (Difficult, Uncontrollable)
S1 LIGHT AND MODERATE INJURIES	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	QM
	E3 (Medium)	QM	QM	A
	E4 (High)	QM	A	B
S2 SEVERE AND LIFE THREATENING INJURIES – SURVIVAL PROBABLE	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	A
	E3 (Medium)	QM	A	B
	E4 (High)	A	B	C
S3 LIFE THREATENING INJURIES, FATAL INJURIES	E1 (Very low)	QM	QM	A
	E2 (Low)	QM	A	B
	E3 (Medium)	A	B	C
	E4 (High)	B	C	D

QM (Quality Management)
Development supported
by established Quality Management
is sufficient.

A B C D

lowest ASIL
Low risk reduction necessary
⋮
highest ASIL
High risk reduction necessary

Figure 11 - ASIL overview

After defining safety goals and their criticality through ASIL, the functional safety concept phase focuses on deriving the specific functional safety requirements. These requirements are then allocated to the preliminary architectural elements of the item or external safety measures. The complete functional safety concept addresses:

- fault detection and mitigation: identifying and responding to system malfunctions to prevent safety goals from being violated;
- transitioning to a safe state: ensuring the system reaches a safe condition following a fault, minimizing potential harm;
- fault tolerance mechanisms: making the system resilient to faults (for instance with redundancy or diversity), ensuring it can maintain a safe state (with or without reduced functionality) despite malfunctions;
- driver warning: alerting the driver to potential issues (e.g., engine malfunction light, ABS warning light) to enable appropriate action and reduce risk exposure.

Next, there is the implementation phase that pursues two parallel paths:

- product development at the hardware level - this path focuses on implementing the safety requirements in the physical hardware components of the system;
- product development at the software level - this path focuses on implementing the safety requirements in the software code of the system.

For each path, the standard provides specific guidance on:

- required methods and measures, i.e. the data to be collected and the methods to be used to demonstrate compliance with the standard and to ensure the effectiveness of the security measures implemented;
- recommended methods and measures, i.e. the appropriate approach to achieve the desired safety level.

The complex process of product development at the system level is further structured into several subphases, as illustrated in the figure:

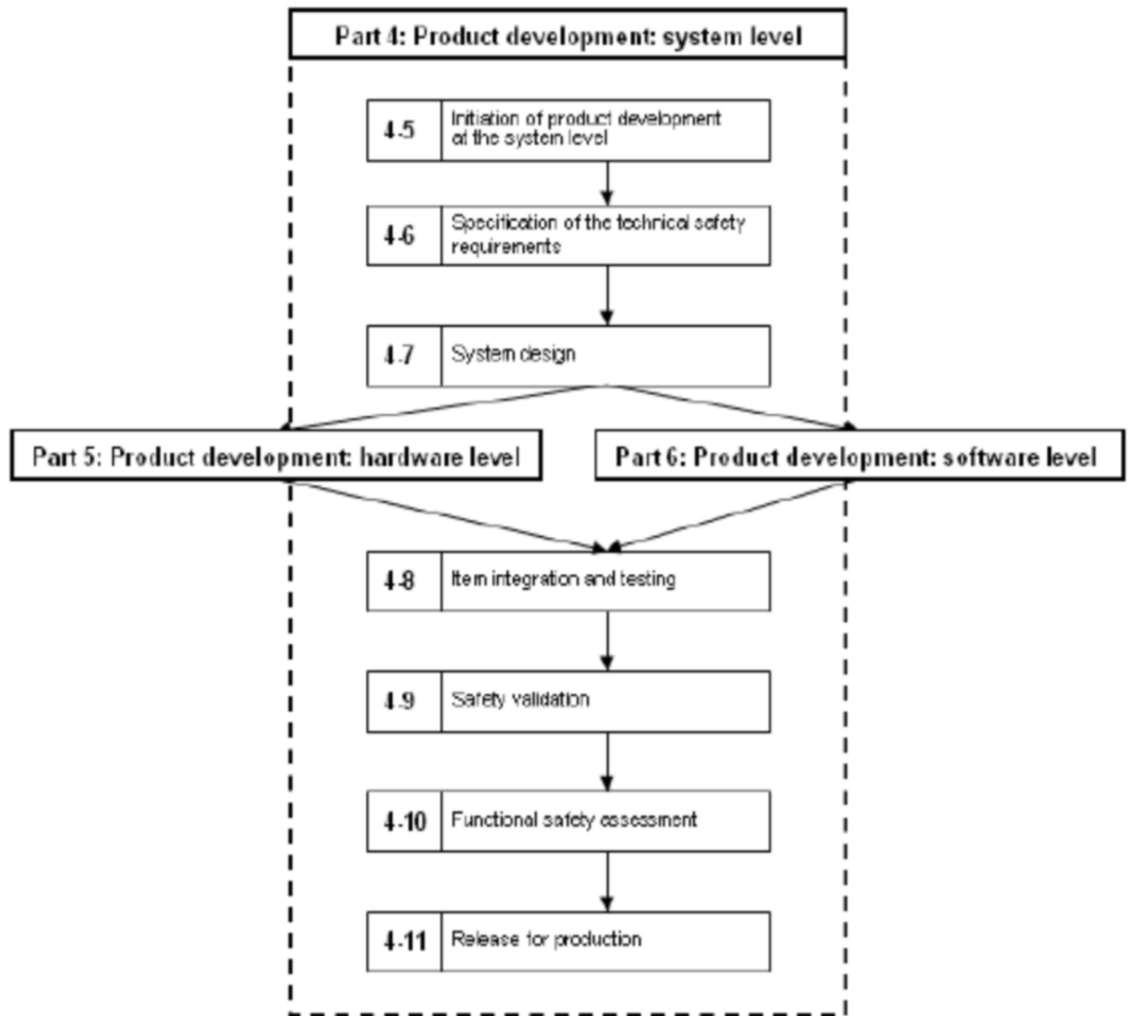


Figure 12 – Product development at the system level phase

Now, let's delve into some of the key steps in this process, along with the essential verifications required for each.

Specification of technical safety requirements

This stage focuses on refining the functional safety concept into specific technical safety requirements. These requirements consider both the functional concept (intended behaviour) and the preliminary architectural assumptions (hypotheses about the system structure).

Verification: following specification, a thorough analysis is conducted to verify that the technical safety requirements effectively fulfil the functional safety requirements. This ensures alignment and adherence to the intended safety goals.

System design

This stage aims to develop a system design and a corresponding technical safety concept that adhere to the established functional requirements and the technical safety requirements specification for the item.

Verification: following the design phase, a rigorous verification process confirms that the final system design and the associated technical safety concept align with the technical safety requirements specification. This ensures the system meets the established safety goals.

HW/SW interaction

This crucial phase ensures coordinated interaction between hardware (HW) and software (SW) components. It involves defining and specifying:

- relevant operating modes and configuration parameters - this outlines the different operational states of hardware devices and their corresponding configuration settings;
- hardware features for independence and partitioning - this focuses on hardware capabilities that facilitate the separation of software elements (partitioning) and ensure their independent operation;
- shared and exclusive use of resources - this defines how hardware resources are allocated, specifying whether they can be used by multiple software components simultaneously or reserved for exclusive use;
- access mechanisms for hardware devices - these detail the methods used by software to interact with hardware devices;
- timing constraints for safety-critical services - this outlines the time limitations imposed on specific services involved in the overall safety concept.

Verification: a thorough analysis is conducted to ensure the technical safety requirements align with the functional safety requirements.

Integration and testing

This phase guarantees compliance with all safety requirements and verifies their effective implementation through the item's design. The integration process consists of three key parts:

- HW/SW level integration - testing the seamless interaction of hardware and software within each element of the item;
- System level integration - testing how the individual elements function together to form the complete item;
- Vehicle level integration - testing how the item interacts with other systems within the vehicle and with the entire vehicle itself.

Verification: by carrying out a rigorous validation process during the process, engineers gain a high level of confidence that the integrated system meets all established safety requirements. This rigorous approach promotes the development of robust and reliable automotive systems that prioritise safety as a critical design principle.

Chapter 3

MODEL-BASED DESIGN

The following chapter focuses on Model-Based Design (MBD), a design method used extensively throughout this project. MBD provides a mathematical and visual approach to the development of complex control and signal processing systems. It focuses on the use of a system model, that reflects all the relevant parts and properties of the system, throughout the entire development chain, from initial design, analysis, and simulation to automatic code generation, and verification. Model-based design moves the design phase from the lab and field to the desktop, resulting in a reduction in costs and time. Instead, Model-Based Software Design (MBSD) is a software development process that addresses the increasing complexity of software development through abstraction and automation. Abstraction is achieved by using appropriate models of a software system, while automation systematically transforms these models into executable source code. Engineers create a model to specify the behaviour of an embedded system; the model, consisting of block diagrams, textual programs, and other graphical elements, is an executable specification that allows engineers to run simulations to test ideas and verify designs throughout the development process. The benefits are the following:

- improve product quality: testing activities during the design and development phase allow to improve the quality of the product;

- developing high complexity functions: classical software development is difficult to use to design functions with high complexity;
- costs and time-saving: simulations are performed on a software model and transformations are used to obtain final artefacts, like executable code.

Model-driven development helps to develop highly complex functions with viewer iterations, resulting in less development effort; in fact, this approach leads to:

- better communication: the graphical nature of the models makes them very useful for communicating with other experts. It's also possible to involve people who are not familiar with software development thanks to the use of models. This helps to include additional know-how in the software development;
- rapid control prototyping: this step reduces development time by allowing corrections to be made early in the product process. This allows errors to be corrected and changes to be made while they are still inexpensive;
- reduce software errors: code can be automatically generated for embedded deployment, saving time, and eliminating the risk of manually introducing errors into the code.

One of the most popular tools for MBSD is Simulink by MathWorks. It is a software integrated with MATLAB for modelling, simulation, and analysis of dynamic systems developed by MathWorks. Instead of manually writing thousands of lines of code, Embedded Coder provides the ability to automatically generate high-quality C, C++, and VHDL code that behaves the same as the model created in Simulink; it extends MATLAB CoderTM and Simulink CoderTM with advanced optimizations for precise control over the generated functions, files, and data [4].

Having established the foundation, we can now proceed to a more specific examination of the MBSD's application, including both the logical sequence and the practical steps

required for its implementation. MBSD offers a structured approach to define system functionality; here's a key aspect of this process:

- the system's core functionality is described within a Platform-Independent Model (PIM), which utilizes a Domain-Specific Language (DSL), specifically designed for the system's domain of operation, capturing the system's essential properties in a clear, concise, and unambiguous manner;
- the PIM then undergoes automatic translation into a Platform-Specific Model (PSM) executable by computers.

The V-model, also known as the V-cycle in the automotive industry, is a widely used software development methodology known for its rigorous approach to managing the entire project lifecycle. It takes its name from the distinct V-shape it forms when the development and testing phases are visualised working in tandem. The V-model divides the process into two main parts: the left part of the V deals with the requirements analysis, and functional/software design, while the right part of the V concentrates on the integration, testing and final verification activities. So, the left side of the model can also be called the *Validation phase*, while the right side can be called the *Verification phase*; the two processes are joined by the *Coding phase* at the base of the V. With this model, each step of the lifecycle has a corresponding test plan that helps identify errors early in the life cycle, minimise future issues, and verify adherence to project specifications. Thus, the V-model lends itself well to proactive defect testing and tracking. However, a drawback of the V-model is that it is rigid and offers little flexibility to adjust the scope of a project. Not only is it difficult, but it is also expensive to reiterate phases within the model [5].

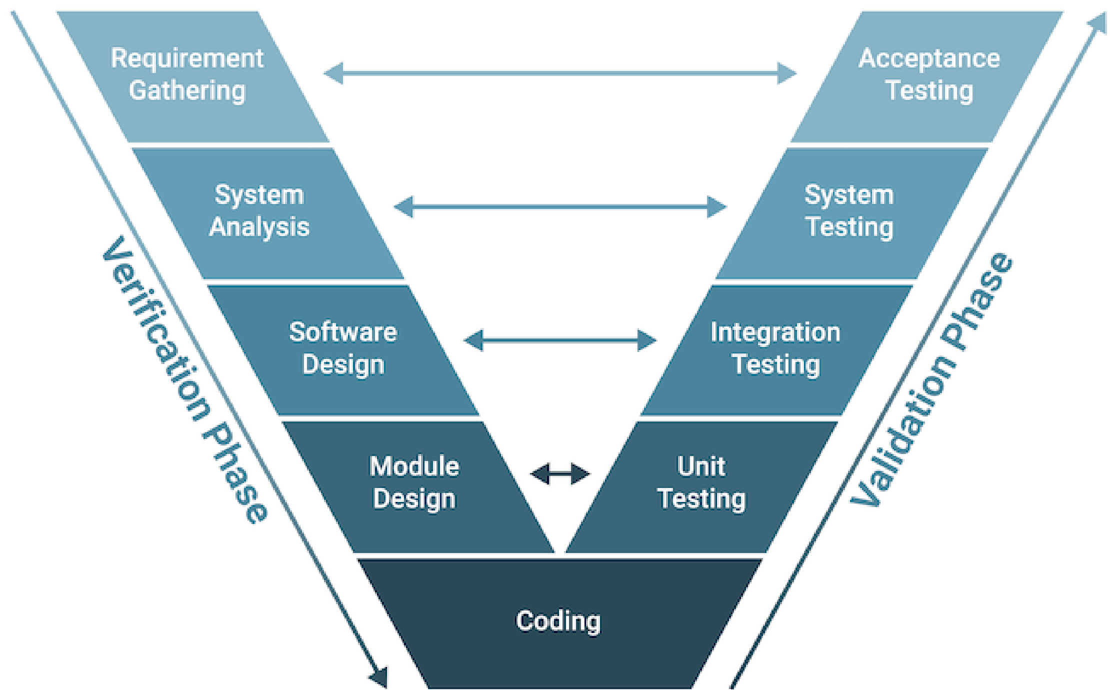


Figure 13 - V-model

Verification phase

The verification phase refers to the practice of evaluating the product development process to ensure the team meets the specified requirements. The phase includes several steps: business requirement analysis, system analysis, software architecture design, and module design:

- in the business requirement analysis step, the team comes to understand the product requirements as laid out by the customer;
- in the system analysis step, the system engineers analyse and interpret the business requirements of the proposed system by studying the user requirements document;
- in the software architecture design stage, the team selects the software architecture based on the list of modules, the brief functionality of each module, the interface

relationships, dependencies, database tables, architecture diagrams, technology details and more. The model testing is developed in this phase;

- in the module design stage, the development team breaks down the system into small modules and specifies the detailed design of each subsystem, called low-level design.

Coding phase

The development team selects a suitable programming language based on the design and product requirements. There are, of course, guidelines and standards for coding, like the MathWorks Advisory Board (MAB) guidelines, and the code will go through many reviews to check its performance.

Validation phase

The validation phase involves dynamic analysis methods and testing to ensure the software product meets the customer's requirements and expectations. This phase includes several stages including unit testing, integration testing, system testing and acceptance testing:

- during the unit testing stage, the team develops and executes unit test plans to identify errors at the code or unit level. This testing happens on the smallest entities, such as program modules, to ensure they function correctly when isolated from the rest of the code;
- the integration testing stage involves executing integration test plans developed during the architectural design step in order to verify that groups created and tested independently can coexist and communicate with each other;
- the system testing stage involves executing system test plans developed during the system design step, which are composed by the client's business team. System testing ensures the team meets the application developer's expectations;
- the acceptance testing step is related to the business requirement analysis part of the V-model and involves testing the software product in the user environment to

identify compatibility issues with the different systems available within the user environment. Acceptance testing also identifies non-functional issues like load and performance defects in the real user environment.

The V-cycle emphasises the importance of testing and quality assurance throughout the entire development process [6]. In particular, the observation of the entire design flow makes it possible to distinguish between what is known as In-the-Loop testing.

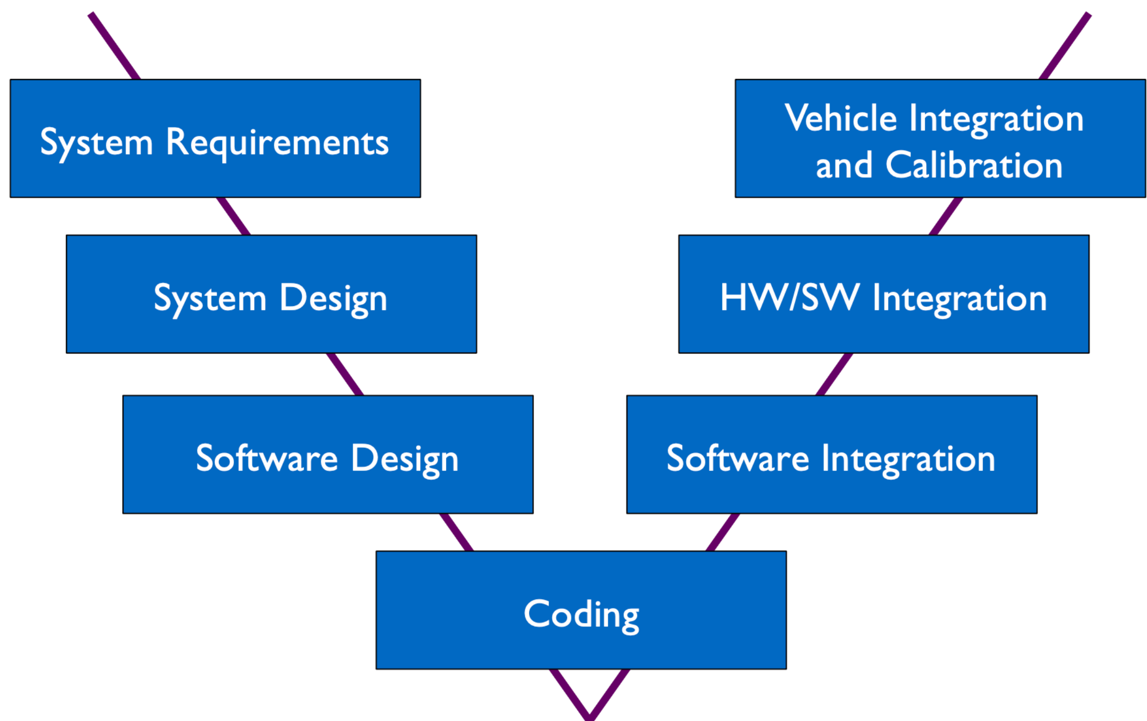


Figure 14 – Design flow diagram

Model-In-the-Loop (MIL) testing

This technique is used to evaluate the functionality of the control algorithm within a simulated environment.

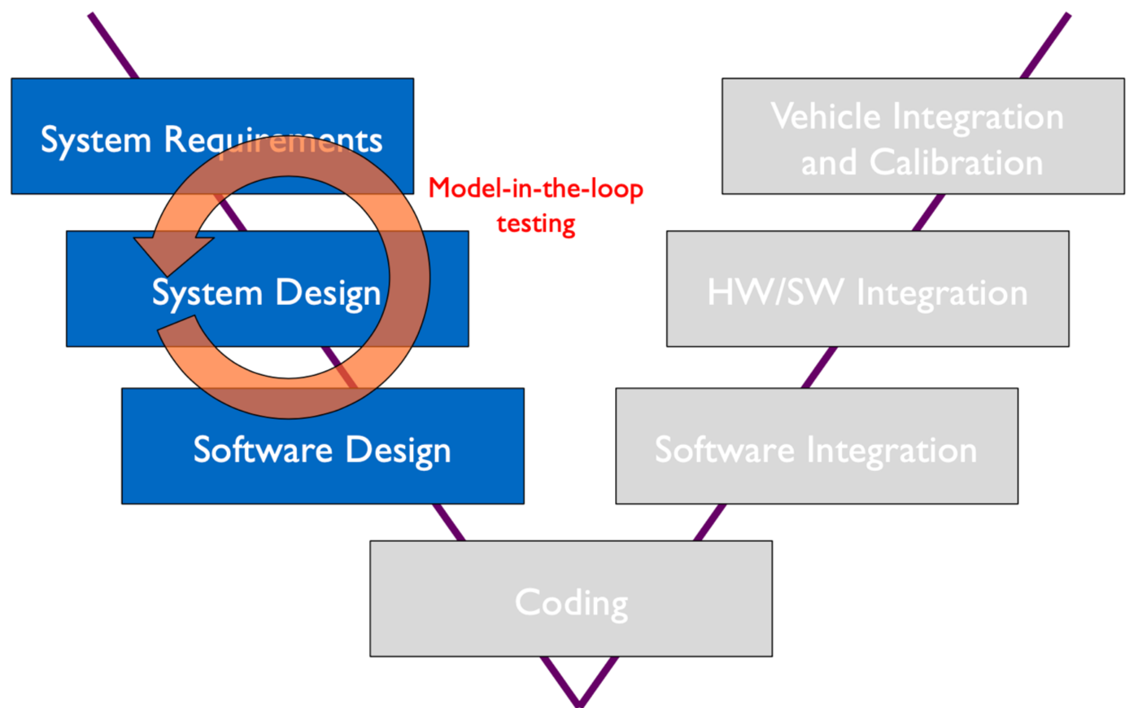


Figure 15 - Model-In-the-Loop (MIL) testing

Both the plant, i.e. the system to be controlled, and the controller, i.e. the algorithm to control the plant, are modelled, so that the model exists entirely in a native simulation tool (e.g. Simulink/Stateflow), and multiple simulations are run to refine the model itself and/or evaluate design alternatives.

Software-In-the-Loop (SIL) testing

Similar to MIL testing, SIL testing evaluates the functionality of the control software in a simulated environment. However, unlike MIL which uses a model of the control algorithm, SIL testing uses the actual embedded software code that has been generated.

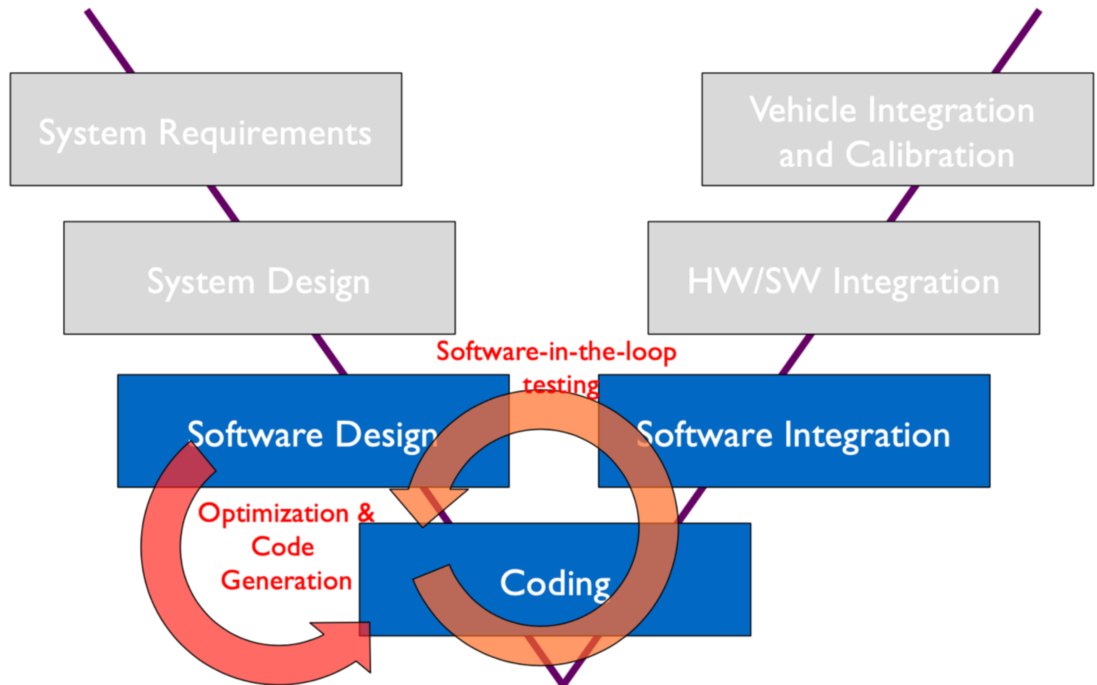


Figure 16 - Software-In-the-Loop (SIL) testing

Part of the model exists in the native simulation tool (e.g. Simulink/Stateflow) and part as executable code, typically in C (e.g. S-function). The controller implementation is co-simulated with the plant model to test its correctness.

Processor-In-the-Loop (PIL) testing

PIL testing is a software development technique based on the concepts of both MIL and SIL testing. This step involves deploying the implementation to the target processor, or a very similar one, such as an Electronic Control Unit (ECU) for rapid prototyping.

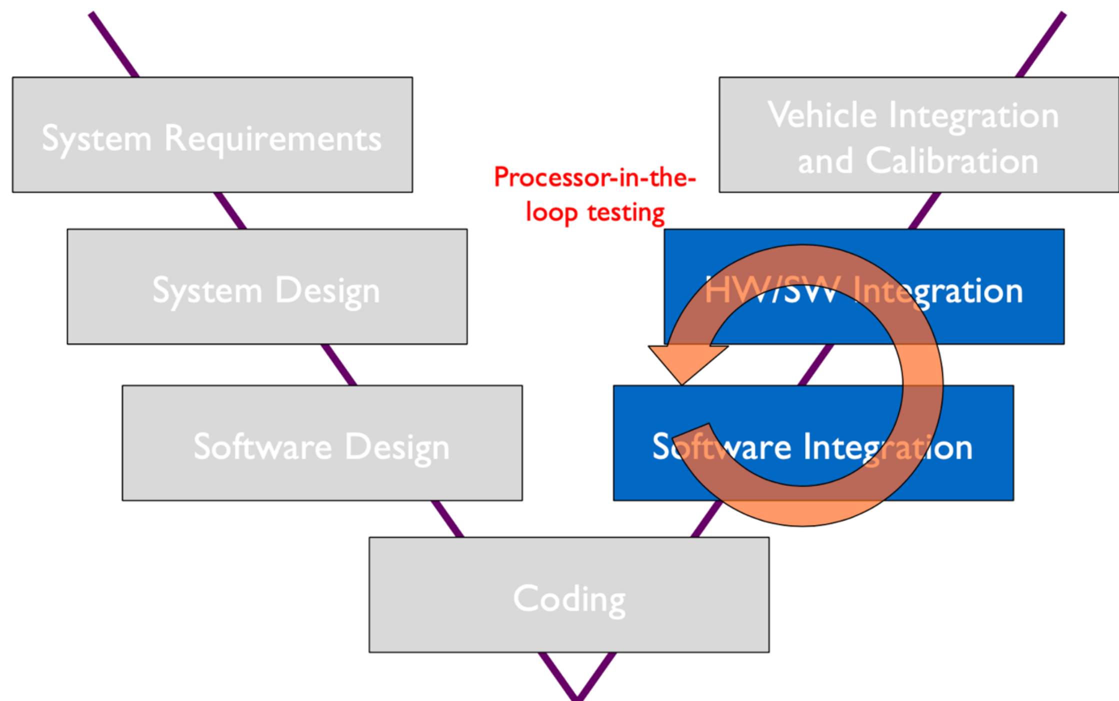


Figure 17 - Processor-In-the-Loop (PIL) testing

Part of the model still exists in the native simulation tool (e.g. Simulink/Stateflow) and part as executable code (e.g. S-function) running on target or rapid prototyping hardware without real-time constraints to fully test and verify the controller implementations in C code.

Hardware-In-the-Loop (HIL) testing

HIL testing is the final critical phase in the development process of complex systems controlled by embedded software. This step involves integrating the actual physical hardware components with an emulated environment representing the system's operational environment. The embedded software code, previously validated through SIL and PIL testing, is loaded onto the hardware components.

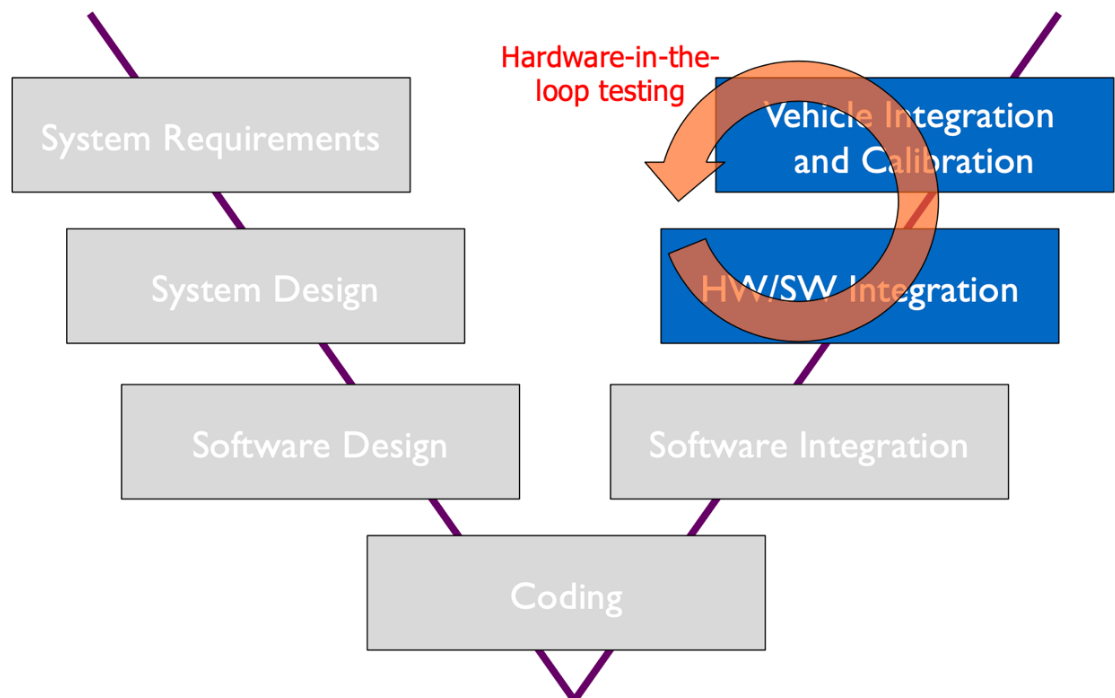


Figure 18 - Hardware-In-the-Loop (HIL) testing

Part of the model runs in a real-time simulator and part exists as physical hardware: the implementation runs on the target hardware (e.g. the ECU), while a model representing the physical system (plant model) runs on separate rapid prototyping hardware. This setup allows real-time testing of the interaction between the software and the emulated plant environment [7].

Chapter 4

REQUIREMENTS AND STATE MACHINE

Having clearly and comprehensively outlined the theoretical basis, it is now possible to deal with what has been done operationally for this thesis. This work hooks up with the EVERGRIN company project, going specifically to re-develop, for academic purposes, the initial phases of the same, always following the V-cycle and model-based design techniques. The process is divided into the following key stages:

- system requirements drafting: the initial step involves defining the main functionalities and requirements of the system;
- state machine creation: a general state machine is then developed to model the system's behaviour under various operational conditions;
- control law development: the control law, governing the system's behaviour, is implemented within the Simulink environment. This allows for simulation and analysis of the control strategy;
- Model-In-the-Loop (MIL) testing: finally, the developed model undergoes rigorous testing to validate its functionality and performance.

This methodical approach, based on the V-cycle, ensures a structured and well-defined development process. The first two phases will be covered in this chapter, while the development of the control algorithm and subsequent testing will be analysed in detail in the next two chapters.

The system under consideration for which the requirements have been written is a battery electric vehicle (BEV). During the initial stage of this project, considerable attention was given to gaining a thorough comprehension of the system's behaviour throughout its complete operational spectrum. This entailed a meticulous examination of all potential key operating scenarios for the car, encompassing both its inactive and active states. For every identified operating condition, a comprehensive analysis was conducted to scrutinise the interactions and behaviour of the crucial components involved. This extensive investigation provided a clear understanding of how the car functions under various circumstances. Furthermore, this phase of the work allowed for the identification of potential challenges or limitations that might arise in different situations. All requirements have been carefully listed according to the latest industry standards, in particular ISO/IEC/IEEE 29148:2018. The standard, titled "Systems and software engineering — Life cycle processes — Requirements engineering" is an international standard that provides guidelines and best practices for requirements engineering in systems and software development. It outlines a structured approach to defining, documenting, and verifying a system's requirements throughout the entire development lifecycle. As stated in the ISO/IEC/IEEE 29148:2018 document, its scope is:

- specifies the required processes implemented in the engineering activities that result in requirements for systems and software products (including services) throughout the life cycle;
- provides guidelines for applying the requirements and requirements-related processes described in ISO/IEC/IEEE 15288 and ISO/IEC/IEEE 12207;

- specifies the required information items produced through the implementation of the requirements processes;
- specifies the required contents of the required information items;
- provides guidelines for the format of the required and related information items [8].

According to the V-cycle, the first step in software modelling is indeed to specify the system requirements, i.e. statements which translate or express a need and its associated constraints and conditions. This phase serves as a crucial foundation for the entire software development process. It outlines the functional and non-functional requirements of the system, capturing the needs and expectations of the stakeholders, i.e. individuals or organizations having a right, share, claim or interest in the system or in its possession of characteristics that meet their needs and expectations. The requirements specification typically includes a detailed description of the software's purpose, its intended users, and the specific features and functionalities it should possess. It also defines any constraints or limitations that need to be considered during the development process. Writing a comprehensive business requirements specification is essential for ensuring that the software development team and the stakeholders are aligned in their understanding of the project. It helps to establish clear communication and provides a reference point for evaluating the success of the final product. Once the requirements specification is complete, the software modelling process can proceed to the next stage, which involves transforming these requirements into a more technical and detailed design. This design phase lays the groundwork for the subsequent steps in the V-cycle, such as implementation, testing, and deployment.

First, it is essential to have a thorough understanding of how the requirements are formulated under the established standard. This involves comprehending the key principles and guidelines that govern the construction of these requirements. By doing so, one can ensure that the resulting requirements are aligned with the standard's

specifications and objectives. According to the standard document, well-formed stakeholder requirements, system requirements and system element requirements shall be developed. This practice contributes to requirements validation with the stakeholders and helps ensure that the requirements accurately capture stakeholder needs. A well-formed specified requirement contains one or more of the following:

- it shall be met or possessed by a system to solve a problem, achieve an objective or address a stakeholder concern;
- it is qualified by measurable conditions;
- it is bounded by constraints;
- it defines the performance of the system when used by a specific stakeholder or the corresponding capability of the system, but not a capability of the user, operator or other stakeholder; and it can be verified (e.g., the realization of the requirement in the system can be demonstrated.

A requirement is a statement that translates or expresses a need and its associated constraints and conditions. A requirement can be written in the form of a natural language or some other form of language. If expressed in the form of a natural language, the statement should include a subject and a verb, together with other elements necessary to adequately express the information content of the requirement. A requirement shall state the subject of the requirement (e.g., the system, the software, etc.), what shall be done (e.g., operate at a power level, provide a field for) or a constraint on the system. Condition-action tables and use cases are other means of capturing requirements. It is important to agree in advance on the specific keywords and terms that signal the presence of a requirement. A common approach is to stipulate the following.

- Requirements are mandatory binding provisions and use 'shall'.

- Non-requirements, such as descriptive text, use verbs such as are', 'is', and 'was'. It is best to avoid using the term 'must, due to potential misinterpretation as a requirement.
- Statements of fact, futurity, or a declaration of purpose are non-mandatory, non-binding provisions and use 'will'. 'Will' can also be used to establish context or limitations of use.
- Preferences or goals are desired, non-mandatory, non-binding provisions and use 'should'. They are not requirements.
- Suggestions or allowances are non-mandatory, non-binding provisions and use 'may'.
- Use positive statements and avoid negative requirements such as 'shall not'.
- Use active voice: avoid using passive voice, such as 'it is required that'.
- Avoid using terms such as 'shall be able to'.

All terms specific to requirements engineering should be formally defined and applied consistently throughout all requirements of the system.

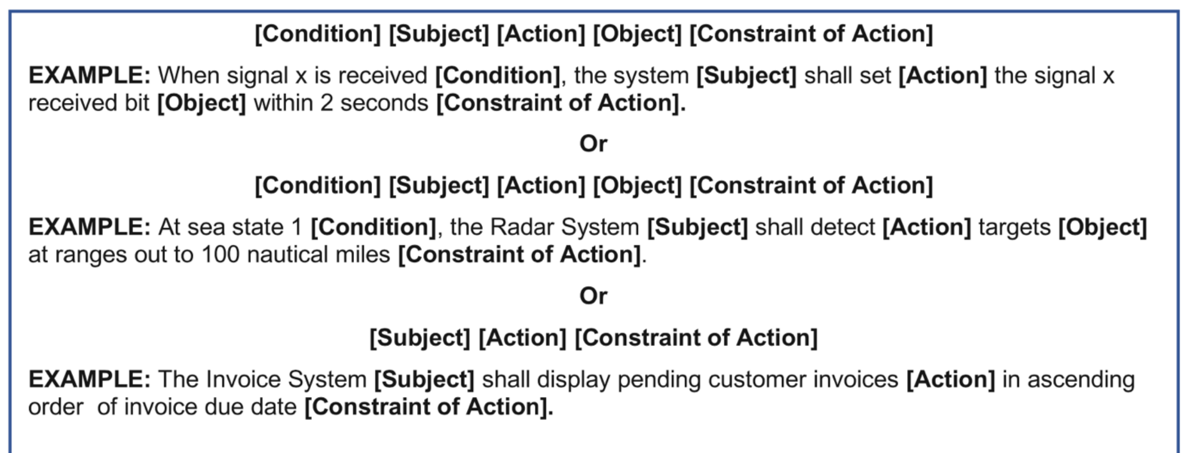


Figure 19 - Examples of functional requirements syntax

Conditions are measurable qualitative or quantitative attributes that are stipulated for a requirement. They further qualify a requirement that is needed and provide attributes that permit a requirement to be formulated and stated in a manner that can be validated and verified. Conditions may limit the options open to a designer. It is important to transform the stakeholder needs into stakeholder requirements without imposing unnecessary bounds on the solution space. Constraints restrict the design solution or implementation of the systems engineering process. Constraints may apply across all requirements, may be specified in relationship to a specific requirement or set of requirements, or may be identified as stand-alone requirements (i.e., not bounding any specific requirement).

Requirements may be ranked or weighted to indicate priority, timing or relative importance. Requirements in scenario form depict the system's action from a user's perspective. Each stakeholder, system and system element requirement shall possess the following characteristics.

- Necessary. The requirement defines an essential capability, characteristic, constraint and/or quality factor. If it is not included in the set of requirements, a deficiency in capability or characteristic will exist, which cannot be fulfilled by implementing other requirements. The requirement is currently applicable and has not been made obsolete by the passage of time. Requirements with planned expiration dates or applicability dates are clearly identified.
- Appropriate. The specific intent and amount of detail of the requirement is appropriate to the level of the entity to which it refers level of abstraction appropriate to the level of entity). This includes avoiding unnecessary constraints on the architecture or design while allowing implementation independence to the extent possible.
- Unambiguous. The requirement is stated in such a way so that it can be interpreted in only one way. The requirement is stated simply and is easy to understand.

- Complete. The requirement sufficiently describes the necessary capability, characteristic, constraint or quality factor to meet the entity need without needing other information to understand the requirement.
- Singular. The requirement states a single capability, characteristic, constraint or quality factor.
- Feasible. The requirement can be realized within system constraints (e.g., cost, schedule, technical with acceptable risk.
- Verifiable. The requirement is structured and worded such that its realization can be proven (verified) to the customer's satisfaction at the level the requirements exist. Verifiability is enhanced when the requirement is measurable.
- Correct. The requirement is an accurate representation of the entity need from which it was transformed.
- Conforming. The individual items conform to an approved standard template and style for writing requirements, when applicable.

In addition, there are also requirement language criteria. When writing textual requirements, implementing the following considerations will result in well-formed requirements employing the characteristics above. Requirements should state 'what' is needed, not 'how'. Requirements should state what is needed for the system-of-interest and not include design decisions for it. However, as requirements are allocated and decomposed through the levels of the system, there can be recognition of design decisions/solution architectures defined at a higher level. This is part of the iterative and recursive application of the requirements, architecture and design processes. Vague and general terms shall be avoided. They result in requirements that are often difficult or even impossible to verify or may allow for multiple interpretations. The following are types of unbounded or ambiguous terms:

- superlatives (such as 'best', 'most');
- subjective language (such as 'user friendly', 'easy to use', 'cost effective'');

- vague pronouns (such as 'it', 'this', 'that');
- ambiguous terms such as adverbs and adjectives (such as 'almost always', 'significant', 'minimal') and ambiguous logical statements (such as 'or', 'and/or');
- open-ended, non-verifiable terms (such as 'provide support', 'but not limited to', 'as a minimum');
- comparative phrases (such as 'better than', 'higher quality');
- loopholes (such as 'if possible', 'as appropriate', 'as applicable'); terms that imply totality (such as 'all', 'always', 'never', and 'every');
- incomplete references (not specifying the reference with its date and version number; not specifying just the applicable parts of the reference to restrict verification work) [8].

Once the objectives, aims, and writing methods have been outlined, it is time to delve into the practical aspect of drafting the general requirements for the system in question, i.e. a Battery Electric Vehicle (BEV). At this stage, it is important to specify the critical features and capabilities that the BEV should have in order to meet the desired objectives. To begin with, it is central to define the functional requirements of the BEV. This includes determining the major characteristics and functions, as well as the essential components and subsystems that make up the vehicle. In addition, factors such as the driver's control input and charging conditions should be considered to ensure optimal usability and comfort for the user. Next, attention should be given to the safety and security aspects of the BEV; this involves establishing the necessary features, such as electronic braking systems and security cameras or battery protection systems respectively. By clearly defining these general requirements, the foundation for the development of a battery electric vehicle can be laid, ensuring that the final product is consistent with the stated goals and specifications. Below are the tables listing all the requirements that have been defined, strictly following the writing methods dictated by ISO/IEC/IEEE 29148:2018 during their drafting.

For the inactive status and general requirements of the vehicle:

ID	Off state & general
C001	When the user turns off the vehicle the system shall engage the electric brake.
C002	When the vehicle is off the powertrain shall be inactive.
C003	When the vehicle is off the steering wheel shall be inactive.
C004	When the vehicle is off the pedals shall be inactive.
C005	When the vehicle is off the charging system shall be inactive.
C006	When the vehicle is off the headlights shall be off.
C007	When the vehicle is off the infotainment shall be off.
C008	When the vehicle is off the service lights shall be off.
C009	When the vehicle is off the vehicle's security cameras shall be on.
C010	When the vehicle is off the vehicle's battery protection system shall be inactive.

Figure 20 – Off state requirements

For the charging phase and related failures:

ID	Charging state & related failure
C011	When from the off state the user plugs the vehicle, the system shall start the charging procedure.
C012	During the charging the powertrain shall be inactive.
C013	During the charging the steering wheel shall be inactive.
C014	During the charging the pedals shall be inactive.
C015	During the charging the charging system shall be active.
C016	During the charging of the vehicle the headlights shall be activable.
C017	During the charging of the vehicle the infotainment shall be activable.
C018	During the charging of the vehicle the service lights shall be activable.
C019	During the charging the vehicle's security cameras shall be active.
C020	During the charging the vehicle's battery protection system shall be active.
C021	When the battery reaches 100% of the charge the system shall interrupt the charge.
C022	If a fault occurs during the charging, the system shall enter in charging fault mode.
C023	If the system resolves the charging faults, the system shall go back to the previous state.

Figure 21 – Charging state requirements

For the start-up process and related failures:

ID	Start-up state & related failure
C024	When from the off state the user pushes once the start button, the vehicle shall begin the start-up procedures.
C025	During the start-up procedure the powertrain shall start the activation procedure.
C026	During the start-up procedure the headlights shall be activable.
C027	During the start-up procedure the infotainment shall be active.
C028	During the start-up procedure the service lights shall be activable.
C029	During the start-up procedure the charging system shall be inactive.
C030	During the start-up procedure the security cameras shall be off.
C031	If a fault occurs during the start-up procedure, the system shall enter in start-up fault mode.
C032	If the system resolves the start-up faults, the system shall go back to the previous state.

Figure 22 – Start-up state requirements

For the traction mode and related failures:

ID	Traction state & related failure
C033	When the start-up is complete the driver shall push once the start button to begin driving.
C034	When in traction the system shall disable the electronic hand brake.
C035	When in traction the headlights shall be activable.
C036	When in traction the infotainment shall be activable.
C037	When in traction the service lights shall be activable.
C038	When in traction the powertrain shall be active.
C039	When in traction the steering wheel shall be active.
C040	When in traction the pedals shall be active.
C041	When in traction the charging system shall be inactive.
C042	If the gear selector is in D and the driver pushes the accelerator pedal, the vehicle shall go forward.
C043	If the gear selector is in D the transmission shall be engaged.
C044	If the gear selector is in R and the driver pushes the accelerator pedal, the vehicle shall go backward.
C045	If the gear selector is in R the transmission shall be engaged.
C046	If the gear selector is in N the transmission shall be disengaged.
C047	If the gear selector is in P the transmission shall be disengaged.
C048	When in traction with the gear selector in P and the driver pushes the start button once, the system shall go off.
C049	If a fault occurs during the driving, the system shall enter in traction fault mode.
C050	If the system resolves the traction faults, the system shall go back to the previous state.

Figure 23 – Traction state requirements

Once all the vehicle requirements have been defined, the next stage involves structuring the creation of a finite state machine. This machine plays a crucial role in describing the macro conditions in which the BEV is involved; it acts as a powerful tool that captures and represents the various states and transitions that the system under study can go through during its operation. By utilizing a finite state machine, engineers can effectively model the different operating scenarios of the vehicle. This includes situations such as starting the vehicle, accelerating, decelerating, and charging. Each of these scenarios can be represented as a state (represented with circles) with transitions (represented with arrows) indicating how the vehicle moves from one state to another. The creation of the machine requires careful consideration of the BEV's capabilities, limitations, and desired functionalities. Once the finite state machine is created, it becomes a valuable tool for analysing and simulating the BEV's behaviour in the primary different scenarios. It allows engineers to identify potential issues, optimise performance, and validate the vehicle's design before physical implementation. The systematic approach employed in developing the state machine for this thesis involved several key

steps. Firstly, a thorough analysis of the system requirements was conducted to identify the necessary macro-states and their corresponding transitions. This analysis helped in defining the overall structure and general layout. Once the initial diagram was created, successive iterations were carried out to refine and enhance the model. Furthermore, as the model was progressively updated, extended, and improved, additional features and functionalities were incorporated. This iterative process allowed for a comprehensive and robust state machine to be developed, capable of accurately representing the desired system behaviour. Throughout the development process, careful attention was given to ensure that the diagram adhered to established design principles and best practices. This approach not only facilitated the creation of a reliable and efficient model but also enabled easy comprehension and maintenance of the state machine by future developers. Overall, the approach employed in developing the state machine for this thesis ensured a well-structured and effective representation of the system's behaviour, providing a solid foundation for further analysis and implementation. The first version of the vehicle state machine provided below already includes the pivotal macro-states and their corresponding basic transitions.

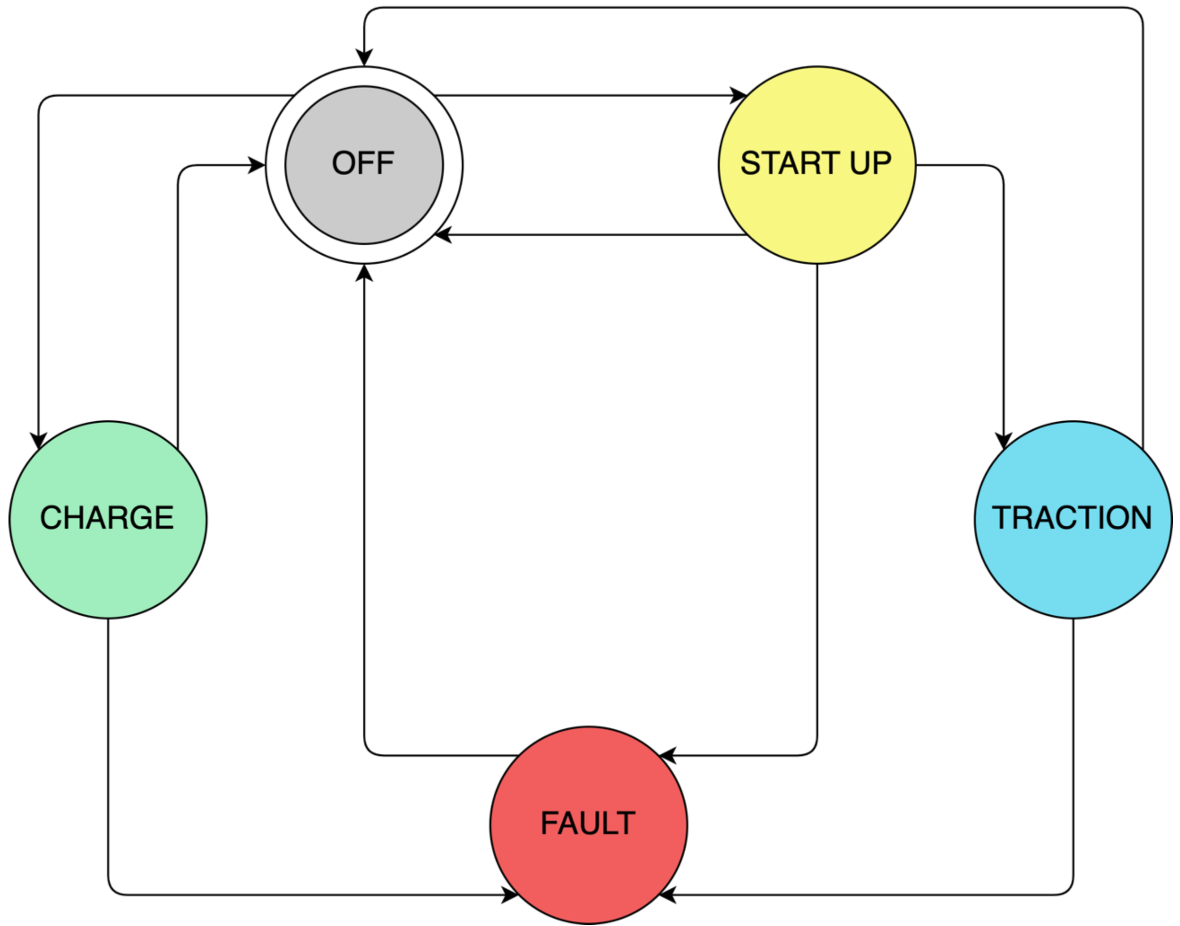


Figure 24 – Version 1 of the vehicle state machine

The cardinal states shown in this diagram are described below:

- *OFF* - the initial state, where the vehicle is powered off, not in use, and parked;
- *START UP* - the state where the vehicle is initializing systems and preparing for driving. This state might include activating headlights or the infotainment system as desired by the user;
- *TRACTION* - the state when the vehicle is fully operational and ready to drive (idle, forward, or backward);
- *CHARGE* - the state when the vehicle is plugged in and undergoing the battery charging process;

- *FAULT* - this general state is accessible from any other state (except for the inactive OFF state) when the system detects a malfunction. A warning is sent to the driver, and some functions, even critical ones, may be disabled.

This initial state machine diagram provides valuable insights into the system it represents. By identifying the key transitions between states, it is possible to visualise the sequential order of execution and the accessibility of each state from others. This information is fundamental for comprehending the system's overall flow and logic. Furthermore, the diagram facilitates a deeper understanding of the system's behaviour. Analysis can reveal how different states interact and how the system transitions between them. This visual representation serves as a powerful tool for grasping the system's structure and functionality, potentially enabling the identification of issues or areas for improvement. The diagram has been improved with the next iteration:

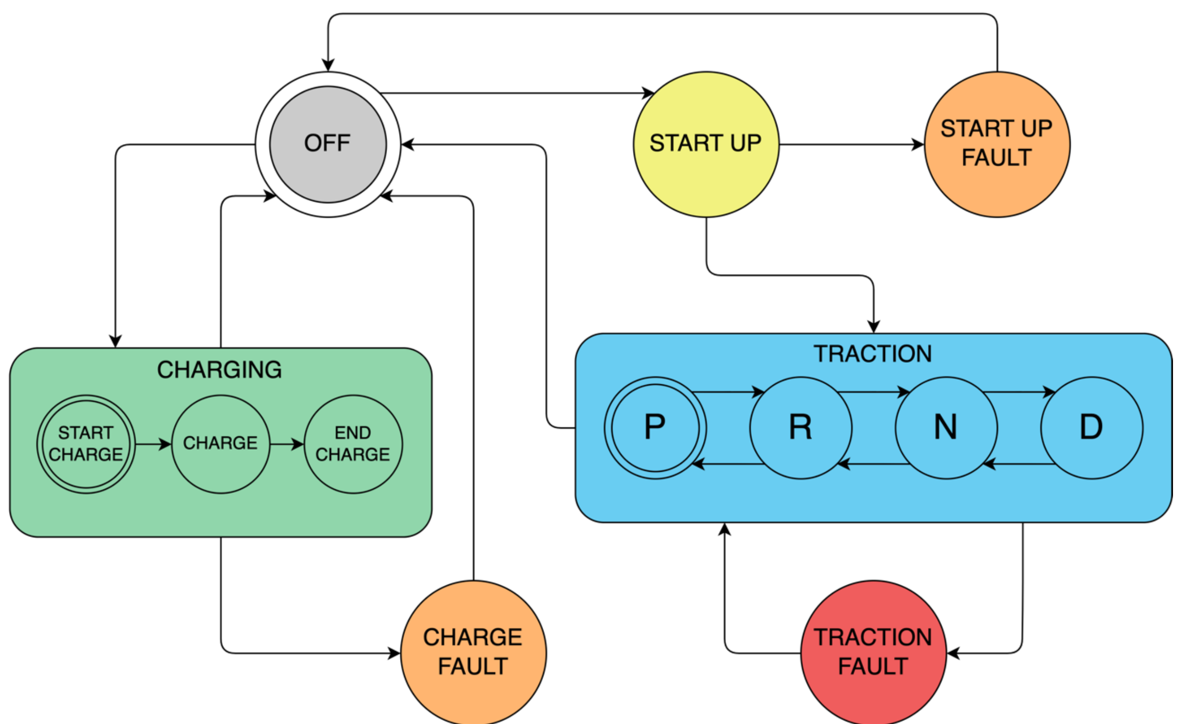


Figure 25 – Version 2 of the vehicle state machine

In this second version, several notable improvements and optimisations have been implemented. Firstly, the modelling of the vehicle's sequential gearbox sub-states has been added to the TRACTION macro-state. This enhancement allows for a more detailed representation of the vehicle's gearbox functionality, providing a more accurate simulation. Additionally, sub-states related to the battery charging process have been incorporated into the CHARGING macro-state. This inclusion enables a more comprehensive depiction of the various stages involved in the battery charging procedure, enhancing the overall realism of the simulation. Furthermore, specific FAULT states have been introduced, depending on the type of fault that occurs and the macro-state in which it happens. This addition allows for a more nuanced representation of potential faults, enabling the simulation to accurately reflect real-world scenarios and their corresponding consequences. Lastly, related transitions and sub-transitions have been included in this version, resulting in a more comprehensive and detailed simulation by capturing the various interactions and processes that occur within the system. Overall, these improvements and optimisations present in the second version enhance the accuracy, realism, and depth of the simulation, providing a more robust and comprehensive representation of the vehicle's functionality and behaviour.

The definitive version of the state machine has been finalized after the third and final iteration. This version will be used to translate the system into the Simulink platform for the upcoming implementation phase:

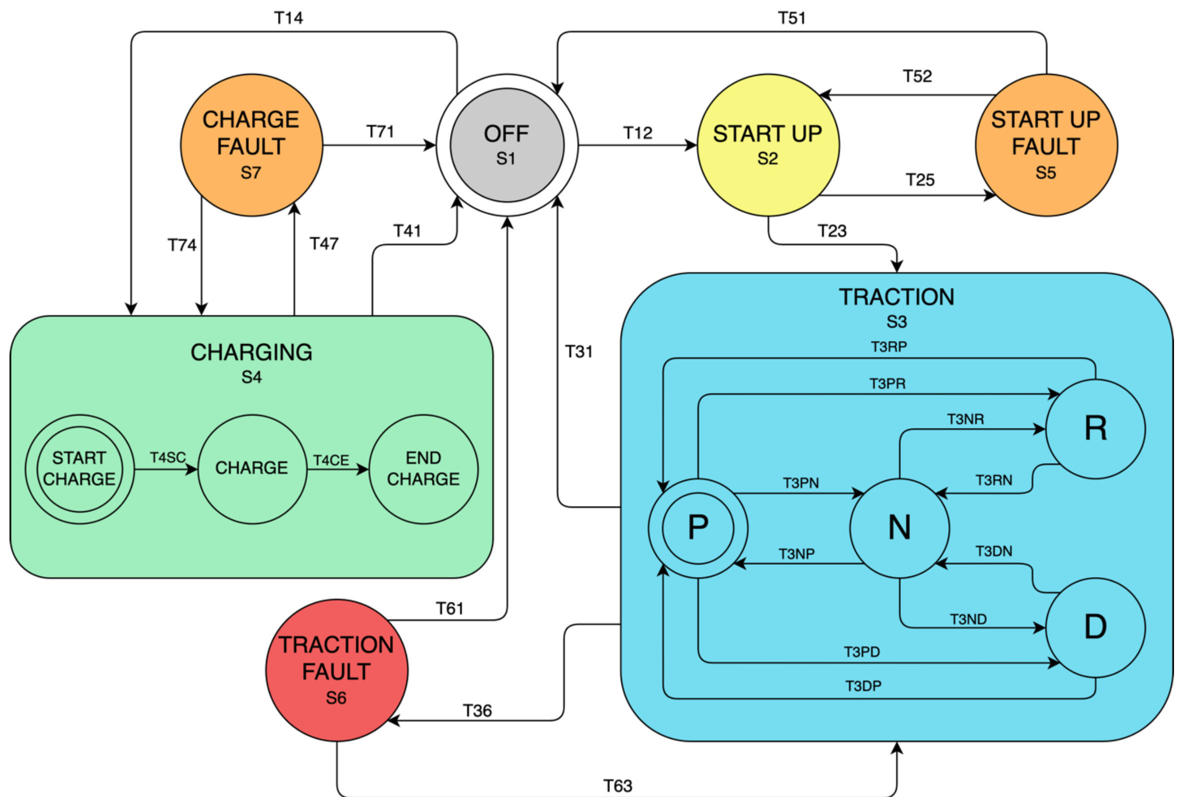


Figure 26 - Version 3 of the vehicle state machine

The changes made to this diagram have greatly improved its functionality and clarity. Firstly, the TRACTION state has been enhanced by incorporating several sub-transitions. This addition allows for a more comprehensive and realistic representation of the model, providing a detailed description of each state within the TRACTION sub-state. Moreover, transitions have been introduced between the macro-states and their corresponding FAULT states. This modification enables the diagram to illustrate the process of returning to the state before the occurrence of a fault, or directly to the off state, once the issue has been resolved. This feature augments the diagram's ability to depict the fault management process accurately. To improve the overall understanding and management of the diagram, labels have been added to each state and transition. These labels serve as markers, making it easier to identify and track the different states and transitions within the diagram. This improvement facilitates efficient management and interpretation of the diagram's different parts. Overall, these implemented changes have significantly

optimised the diagram's level of detail, realism, fault management capabilities, and global usability. With the definition and development of the state machine complete, and the system design complete, the next step was to translate everything into the Simulink environment using the V-cycle methodology. The upcoming chapter will provide comprehensive details about the implementation process, providing a deeper understanding of how the state machine was integrated into the Simulink environment. It will also highlight the specific steps taken and any challenges encountered along the way.

Chapter 5

SOFTWARE IMPLEMENTATION

Once the requirements have been established and the design of the system in question has been outlined, the next phase in the MBD approach is to demonstrate the relevant software implementation. This involves translating the system design into a software model that can be simulated in real time, showing the key aspects of the vehicle and demonstrating the goodness of the previous steps. To achieve this, MBSD techniques utilise various software tools and frameworks that then facilitate the generation of code directly from the system model. Simulink, developed by MathWorks, is a widely used tool in the field of system modelling and simulation. It offers a user-friendly visual interface that allows developers to define the behaviour of a system using various graphical notations, including block diagrams and state machines. With Simulink, engineers can easily design, simulate, and analyse complex systems, making it an ideal choice for this thesis. The tool provides a comprehensive library of pre-built blocks and components, enabling developers to quickly assemble and connect different elements to create a functional system model. Additionally, Simulink supports various mathematical operations and algorithms, making it suitable for a wide range of applications, from control systems to signal processing. Its intuitive interface and powerful simulation

capabilities make Simulink a valuable tool for researchers and engineers alike. Once the system model is complete, it is also possible to automatically generate the corresponding code, which can later be compiled and deployed on the specific target hardware. This code generation process ensures that the software implementation accurately reflects the design specifications, reducing the chances of errors or inconsistencies. In addition, the MBSD procedure encourages a highly iterative and collaborative development process. As the software implementation progresses, developers can continuously validate and verify the behaviour of the system against the established requirements through multiple simulations of the model itself. This iterative approach allows early detection and resolution of any design flaws or functional issues, leading to a more robust and reliable end product. Overall, the use of MBSD techniques streamlines the software development process by providing a systematic and efficient approach to translating system designs into working software implementations. The first step for software implementation was to transpose the completed design directly into the Simulink environment using the integrated Stateflow tool to configure the various states and transitions within the state machine; this process allowed for the precise definition of the system's behaviour and logic. In Simulink, the Stateflow tool provided a visual representation of the state machine, making it easier to understand and modify. Engineers could easily add, remove, or modify states and transitions, ensuring that the system accurately reflected the desired functionality. Additionally, the Stateflow tool offered a range of powerful features to enhance the state machine's functionality. For example, engineers could incorporate conditions and actions within each state and transition, enabling the system to respond to specific events or inputs. This flexibility grants the creation of complex and dynamic state machines that could adapt to evolving conditions. Furthermore, the Simulink environment provided a comprehensive set of simulation and analysis tools. Engineers could simulate, in real time or not, the state machine's behaviour under different scenarios, ensuring that it operated as intended. They could also perform various analyses, such as checking for deadlocks or excessive transitions, to validate the correctness of the state machine. Therefore, the integration of the state machine into the Simulink environment using the Stateflow extension tool facilitated the development and analysis of complex systems,

providing engineers with a powerful toolset to design and validate their designs. The Stateflow implementation of the system state machine is shown below:

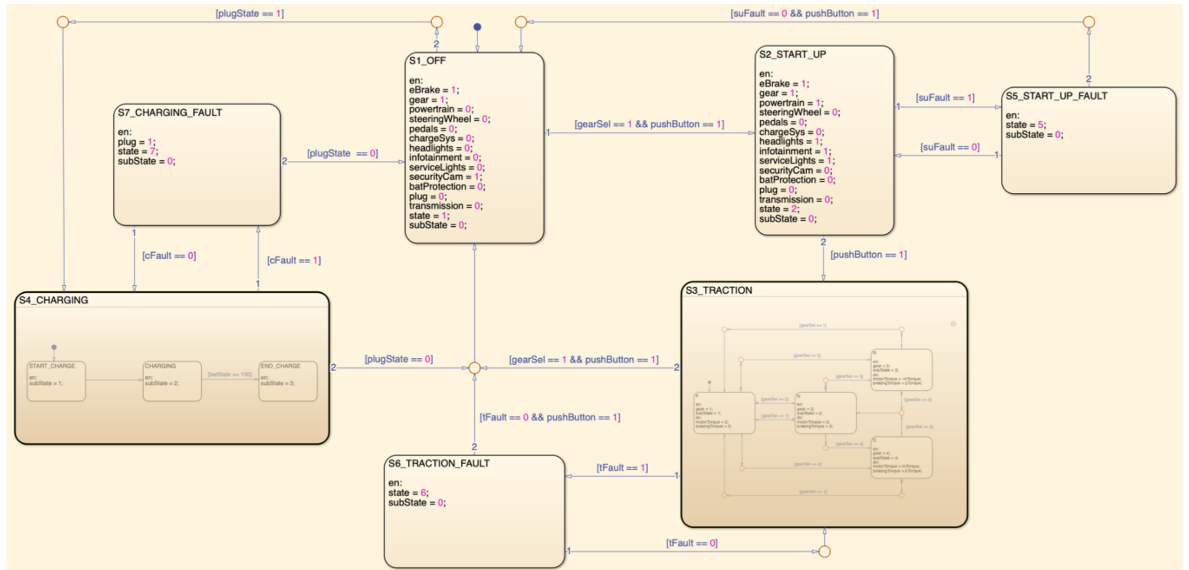


Figure 27 - Stateflow state machine chart

The chart offers valuable insights into the system's behaviour, and a closer examination reveals several key considerations.

Macro-states and system representation

The diagram utilizes rectangles of varying sizes to represent macro-states, which capture the broad operational modes of the system. These macro-states encapsulate the system's behaviour at a high level, allowing for a clear understanding of its overall functioning.

Variables declaration and initialization

For each defined state, all relevant quantities (parameters, variables) are meticulously defined and initialized. This ensures that the state machine has a well-defined starting point for each operational scenario. Notably, for each component within the system, a dedicated boolean variable is declared. This variable acts as a flag, indicating whether the corresponding subsystem is active or inactive in a particular state based on the system

requirements. By incorporating these subsystem activity flags, the state machine explicitly models the behaviour of each component within different operational modes.

Transitions and model clarity

Transitions between states are represented by directional arrows. These arrows depict the flow of the system as it progresses from one state to another based on specific conditions. In the case of complex transition paths, the diagram adheres to MathWorks Advisory Board (MAB) guidelines. This means that right-angled arrows and junctions are used to maintain clarity and avoid cluttered lines, especially when multiple transitions originate or terminate at a single state.

Dynamic data management and state changes

As previously mentioned, variables play a crucial role in the state machine's functionality. These defined variables provide a dynamic mechanism for storing and managing parameters specific to each state. This allows the system to adjust its actions based on changing values within the state. Transitions, on the other hand, define the conditions under which the state machine changes states. They are typically triggered by events (external signals or internal events) and rely on boolean expressions involving variable values. If both the triggering event occurs and the condition evaluates to true, the transition fires and the state machine moves to the designated target state.

State tracking and hierarchy

The state machine incorporates two additional variables, *state* and *subState*; these variables serve as a tracking mechanism, helping to maintain a clear record of the current state and any potential sub-states within a macro-state. This becomes particularly important when dealing with hierarchical state machines, where states can further decompose into a series of smaller, more detailed sub-states. For example, in this model, both the TRACTION and CHARGING macro-states encapsulate additional internal states representing sub-operations within these modes. The *subState* variable would be

crucial for tracking the system's behaviour within these further refined operational categories.

Overall, the state machine diagram effectively combines clear visual representation with well-defined state variables and transitions. This comprehensive approach provides a robust and well-structured model for understanding the system's functionality and interactions among its various components.

Having a complete view of the model structure, excluding the transient fault states, serves as a valuable base for a detailed examination of the main operating states of the system. Fault states, while relevant for safety and robustness, represent exceptional circumstances triggered by failures. By focusing on the primary operating cases within the baseline system behaviour, a more granular analysis is possible.

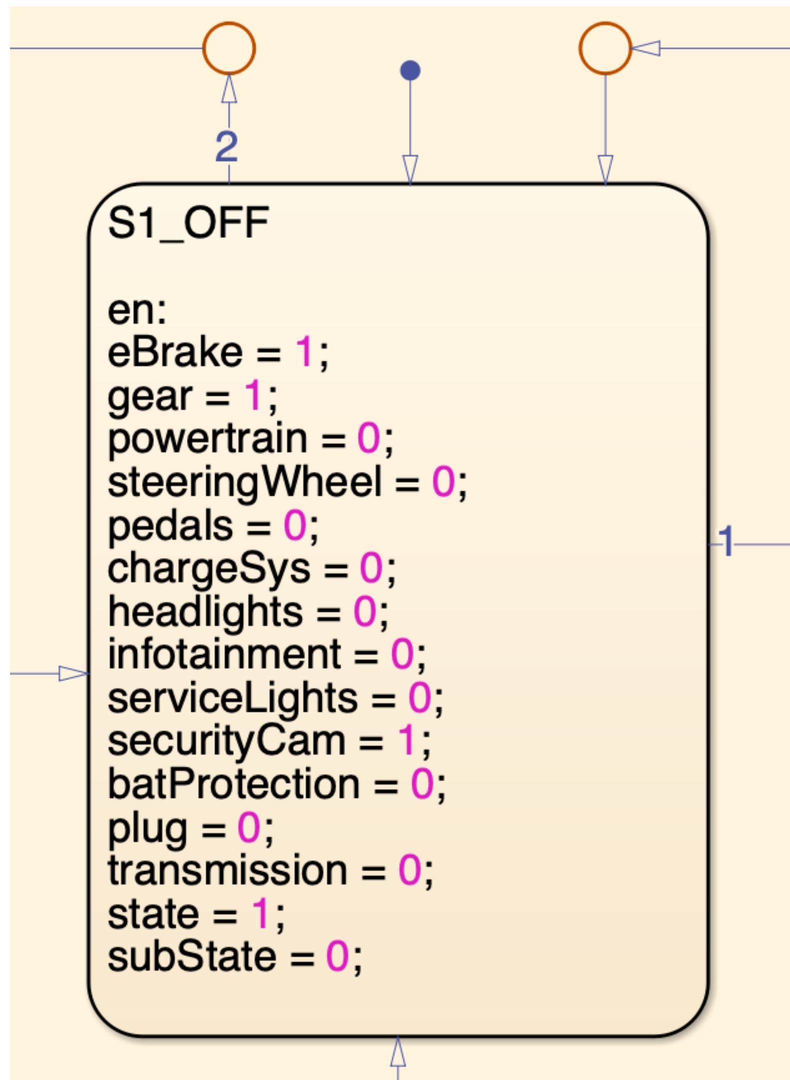


Figure 28 – OFF state

The OFF state plays a crucial role in system initialization. When the simulation is run, the system automatically enters the OFF state as the default state. This state ensures that all the necessary components, from the electronic brake to the powertrain and transmission, are properly settled and ready for operation. Furthermore, the figure provides a comprehensive overview of the system's initial configuration; it shows not only the parameters that have been declared and initialised, but also, like all the other states, the two variables *state* and *subState*, which are essential for monitoring and controlling the current state of the system during operation.

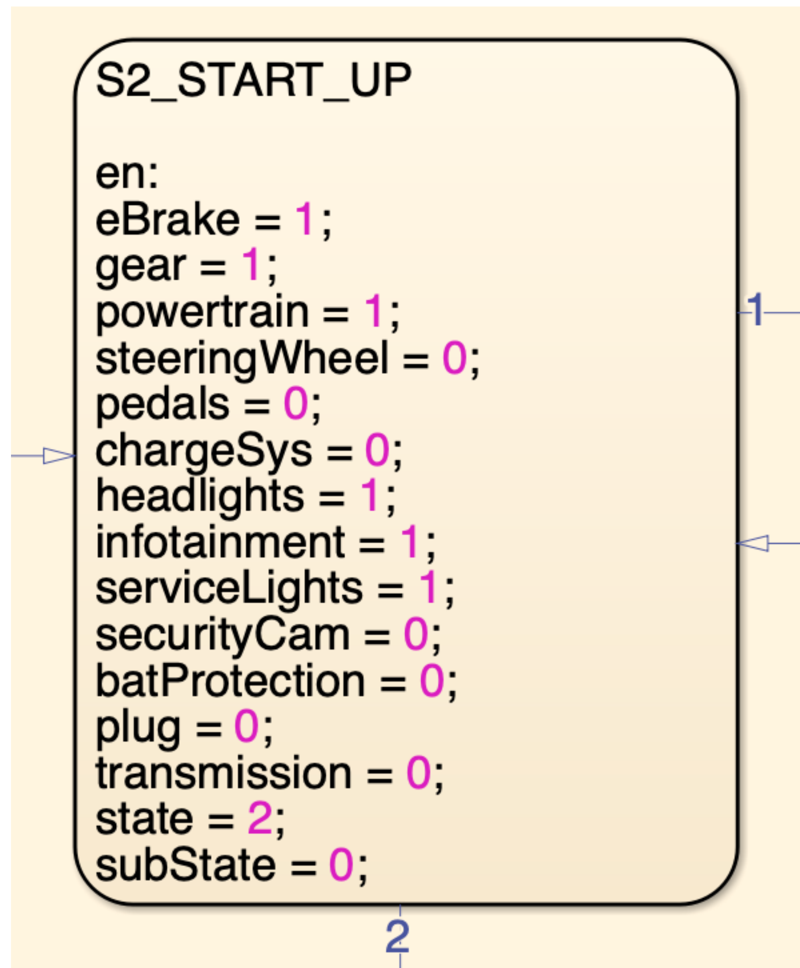


Figure 29 – START UP state

The START UP state is an important condition that allows the vehicle to prepare itself for the next phase of driving. To enter this state, the vehicle must be in the correct gear (P) and the user needs to press the START/STOP button. Once these conditions are met, the vehicle undergoes a series of changes in the values of its boolean parameters. During the START UP state, most of the vehicle's components are activated, ensuring that everything is ready for a smooth driving experience. This includes systems such as the powertrain, transmission, and various safety features. The vehicle's onboard computer checks for any faults or malfunctions before proceeding to the next phase. If any faults or malfunctions are detected during the start-up process, the vehicle shall not proceed to the next driving phase but shall enter the corresponding fault condition, START UP FAULT

state, until the problems have been resolved, if possible. This is an important safety measure to ensure that the vehicle operates optimally and minimizes the risk of any potential problems on the road.

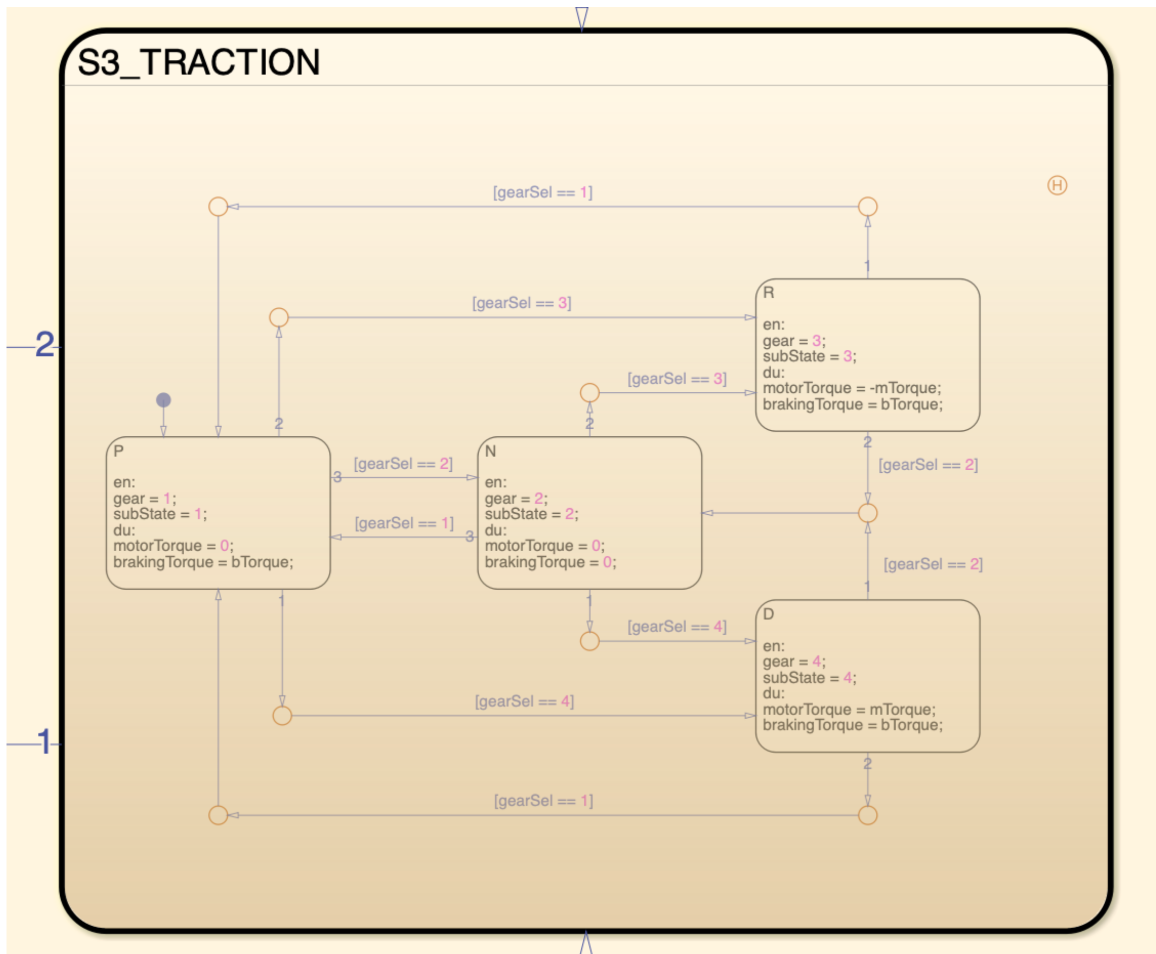


Figure 30 – TRACTION state

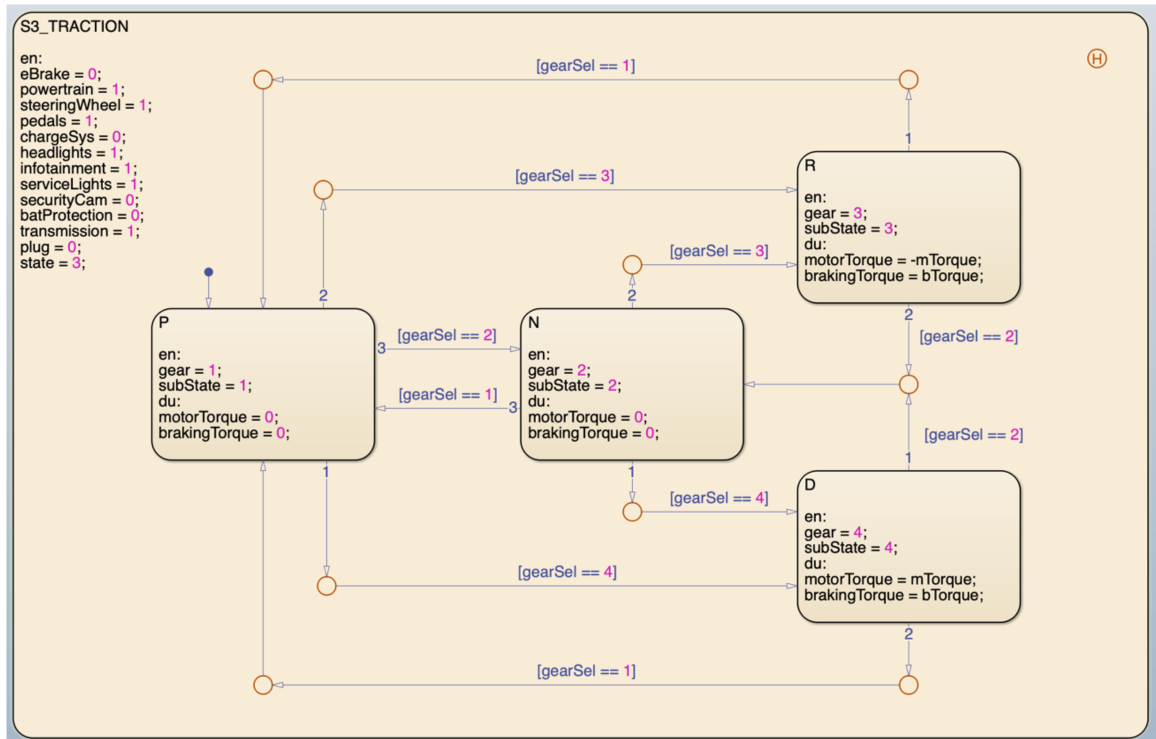


Figure 31 - Insight into the TRACTION state

The TRACTION state is the third non-fault state, and it is a critical one in the system. It encompasses additional sub-states that enhance the accuracy and representation of the sequential shift process. Within the TRACTION state, it is possible to observe the dynamic evolution of all the parameters involved. This allows for a seamless transition from one gear to another, based on the gear selected by the user while driving. Furthermore, two new variables, *motorTorque* and *brakingTorque*, come into play: they fluctuate in response to the driver's acceleration and braking, directly influencing the car's cruising speed. An interesting aspect of the TRACTION state is the inclusion of a history junction in the Stateflow chart. This feature enables the system to keep track of previous sub-states and their corresponding transitions, providing valuable context for the current state of the system.

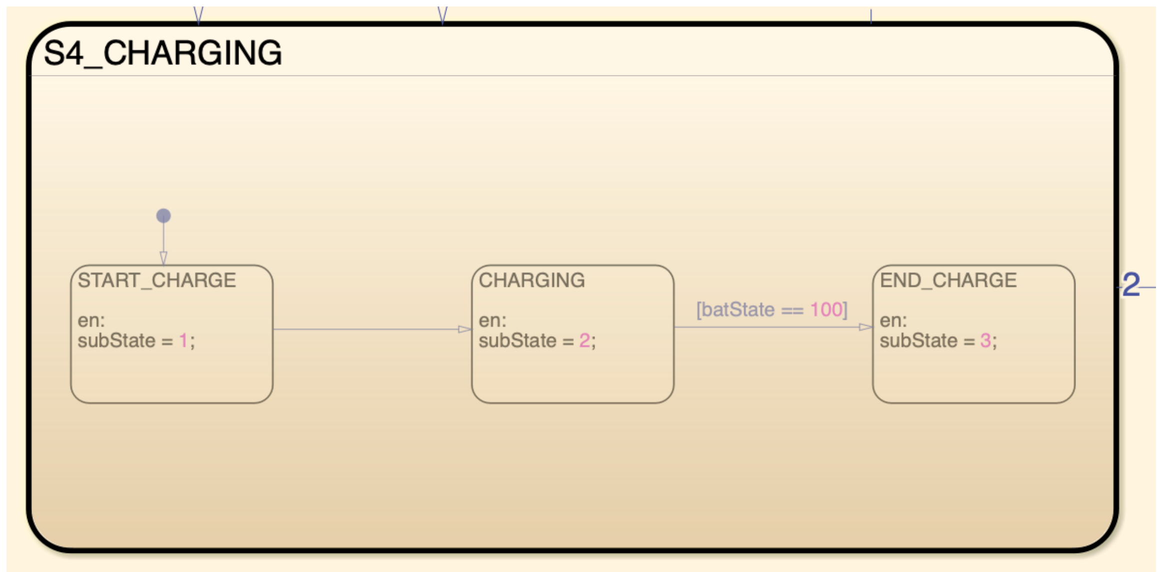


Figure 32 - CHARGING state

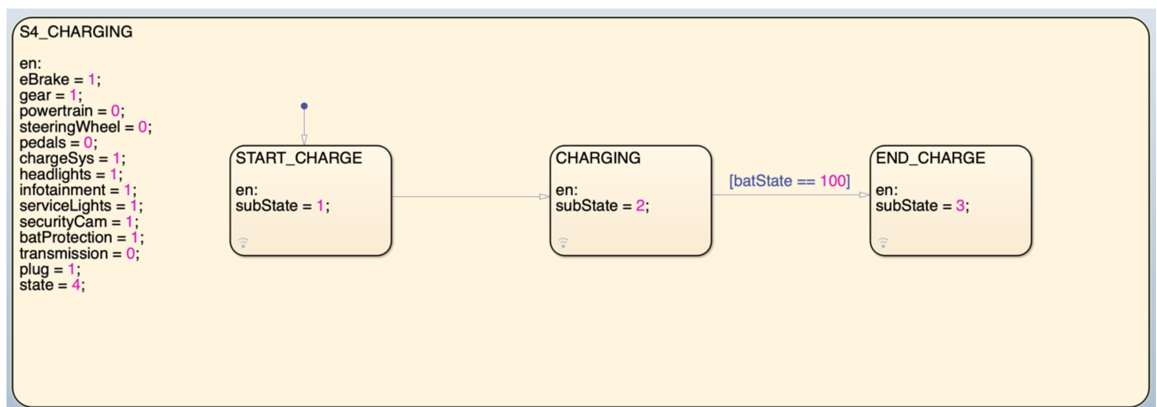


Figure 33 - Insight into the CHARGING state

The last condition examined is the CHARGING condition. This mode is only accessible from the OFF state, i.e. when the vehicle is stationary and parked, and the driver plugs the car. It's important to note that the charging system is designed to protect the battery and other components. Various protections are in place to ensure safe and efficient charging. These protections include monitoring the battery's charge level and temperature, as well as controlling the charging current and voltage. When in the charging scenario, all the components of the charging system, along with their associated

protections, are activated. This means that the variables related to these components are set to 1, indicating their active state. Charging will cease if the user disconnects the vehicle from the power supply or when the battery reaches its maximum charge capacity, as indicated by the sub-states present. This ensures that the battery is not overcharged, which could potentially damage it.

Finally, as mentioned above, in the event of a malfunction, the system immediately switches to the relevant fault state. Apart from the OFF state, which doesn't have a corresponding fault state, there are a total of three fault conditions in the state machine: START UP FAULT, TRACTION FAULT, and CHARGING FAULT. Each of these states serves a specific purpose in identifying and addressing the issues within the system. When the system enters one of these fault states, it ensures that the user is promptly alerted through audible or visual warnings displayed on the dashboard. These warnings serve to inform the driver about the specific fault that has occurred. Additionally, the system may partially reduce the vehicle's functionality to prevent any further damage or risks. While in the compromised state, the vehicle remains in the fault state until the current problems are resolved. Once the issues have been addressed and the nominal state has been restored, it is possible to return to the state prior to the fault or to switch the vehicle directly to the OFF state, based on the drivers' desires or needs.

Chapter 6

SIMULATION AND TESTING

This chapter focuses on the simulation aspect of the high-level vehicle model that has been generated. The primary objective is to provide a detailed description of the simulation framework, the tests performed, and the subsequent analysis of the data obtained. By delving into the simulation process, a comprehensive understanding of how the vehicle model behaves in the different scenarios and conditions will be provided. The tests carried out will cover a wide range of factors according to the requirements provided, allowing a thorough evaluation of the model's capabilities. Analysis of the data collected will provide valuable insights enabling engineers to make informed decisions and improvements. To delve deeper into the simulation, it is essential to provide a complete overview of the developments made in the Simulink environment. Initially, the Stateflow state machine was encapsulated within a subsystem. This encapsulation allowed for seamless integration and interaction with other systems and quantities within the Simulink framework. By encapsulating the state machine, it became possible to establish connections and facilitate communication between the state machine and various components in the Simulink environment. This integration of the Stateflow state machine as a subsystem within Simulink provided a structured and modular approach to the simulation. It enabled the state machine to interact with other blocks, such as sensors, actuators, and controllers, thereby creating a cohesive and interconnected simulation

model. By encapsulating the state machine within a subsystem, it became easier to manage and control the simulation's behaviour. The subsystem acted as a container for the state machine, allowing for efficient organisation and simplifying the overall simulation design. Moreover, this encapsulation facilitated the reuse of the state machine in different simulation scenarios; it could be easily replicated and integrated into other Simulink models, promoting code reusability and reducing development time. In summary, this integration enhanced the simulation's modularity, facilitated communication between different components, and allowed for the efficient management of the simulation's behaviour.

The implementation of the system in the Simulink environment for this thesis aims to achieve a significant objective: the creation of an interactive and real-time simulation. This simulation will be designed to allow users to manage and make modifications to it while it is running, all through a custom-made dashboard. This feature will provide users with a seamless and dynamic experience, enabling them to monitor and control the simulation in real time, making adjustments as needed. The custom dashboard will serve as a central hub, providing a user-friendly interface for easy navigation and efficient management of the simulation. This interactive and real-time capability will enhance the overall effectiveness and flexibility of the system, allowing for better analysis, testing, and experimentation. In the next image is shown the overview of the entire Simulink project:

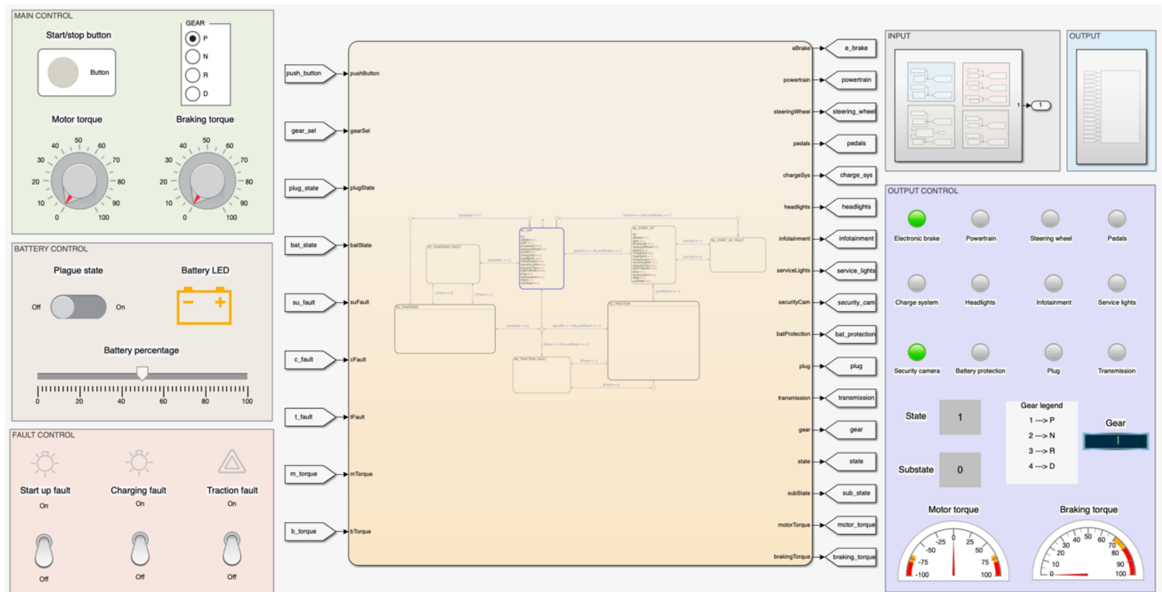


Figure 34 - Overview of the Simulink project

As can be seen, the complexity of the simulation is evident due to its numerous components. One of the key aspects is the management of input and output signals, which are essential for the Statflow state machine. To accomplish this, Simulink Goto and From blocks are employed. These blocks effectively handle the required and generated signals, resulting in a simulation interface that is more streamlined and organised. By eliminating the need for lengthy and convoluted connections between input/output variables and the relevant blocks, this approach aligns with modern development practices and adheres to the reference guidelines. Turning to the dashboard is made up of several panels, each of which encapsulates several functions related to each other or to the same state. Starting on the right side, there is the MAIN CONTROL panel.

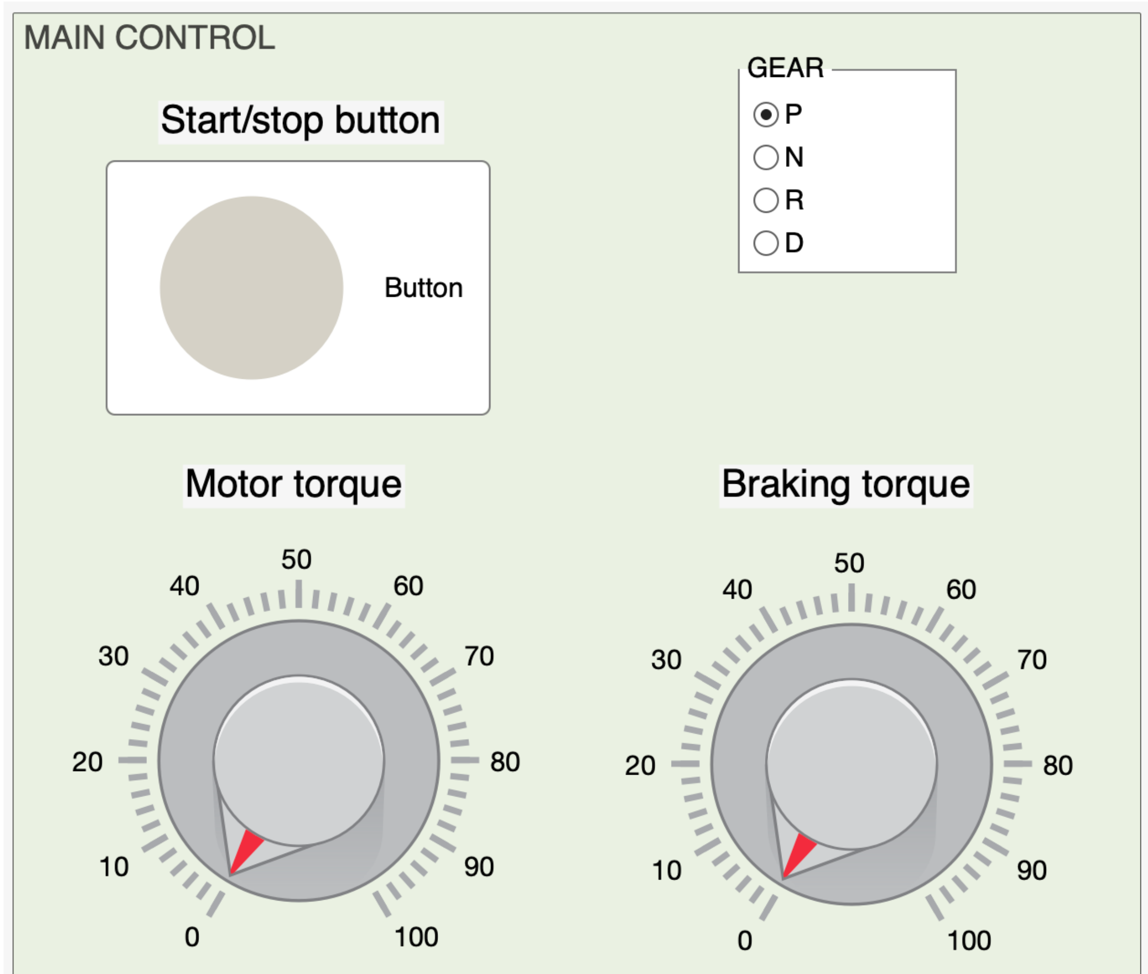


Figure 35 - MAIN CONTROL panel

As the name suggests, the MAIN CONTROL panel serves as a critical user interface element within the real-time simulation. This panel replicates the physical controls found in a real vehicle, enabling users to interact with the simulated system and perform driving manoeuvres. A meticulous examination of these Simulink blocks reveals their functionalities and significance in the context of the overall simulation.

START/STOP button

The top right corner of the control panel houses the START/STOP button block. This block emulates the physical push-button switch commonly found in modern vehicles. Its

primary function lies in initiating the simulated vehicle's operation. When activated by the user, the START/STOP button block transmits a digital signal that triggers the transition from the default OFF state to a non-faulty operational state, excluding the charging state. This action effectively serves as the catalyst for the simulated vehicle to become operational and ready for further user interaction.

Motor torque knob

Located counterclockwise from the START/STOP button lies the motor torque knob block. This block functions as the central element in simulating the driver's interaction with the accelerator pedal. By turning this knob from a scale of 0 to 100 (Newton-metres), the user transmits to the system the signal representing the desired engine torque. In essence, this simulated action dictates the amount of power the system needs to generate for forward or backward motion. Positive torque values correspond to forward propulsion, while negative values correspond to backward propulsion (the electric machine rotates in the opposite sense). This simulation element therefore plays a vital role in providing realistic control of the vehicle's speed and direction.

Braking torque knob

Directly to the right of the motor torque knob is the brake torque knob block. This block acts as a virtual representation of the physical brake pedal. Similar to its counterpart, the application of braking force by the driver is simulated by rotating this knob on a scale from 0 to 100 (Newton metres). The corresponding signal transmitted by the block informs the system of the desired braking torque, ultimately leading to a simulated deceleration of the vehicle. Notably, the simulation offers the potential for future integration of regenerative braking, a technology enabled by Battery Electric Vehicles (BEVs). This innovative feature harnesses the vehicle's kinetic energy during braking: when the driver applies the brakes, the electric motor essentially acts as a generator, converting the kinetic energy back into electrical energy. This recovered energy is then stored in the vehicle's battery, increasing the overall efficiency of the BEV. This inclusion of a realistic braking mechanism adds another layer of fidelity to the simulation.

Gear radio Button

Completing the control panel interface is the gear radio button block. This element mimics the physical gear selector found in traditional vehicles. By selecting the appropriate option on this virtual gear selector, the user transmits a digital signal to the simulated engine control electronics. This signal instructs the system on the desired direction of travel, forward or reverse. This crucial element empowers the user to control the vehicle's direction within the simulated environment, adding another layer of realism and enabling manoeuvring capabilities.

The second panel, located immediately below the MAIN CONTROL panel, is the BATTERY CONTROL panel, used to manage the charging state.

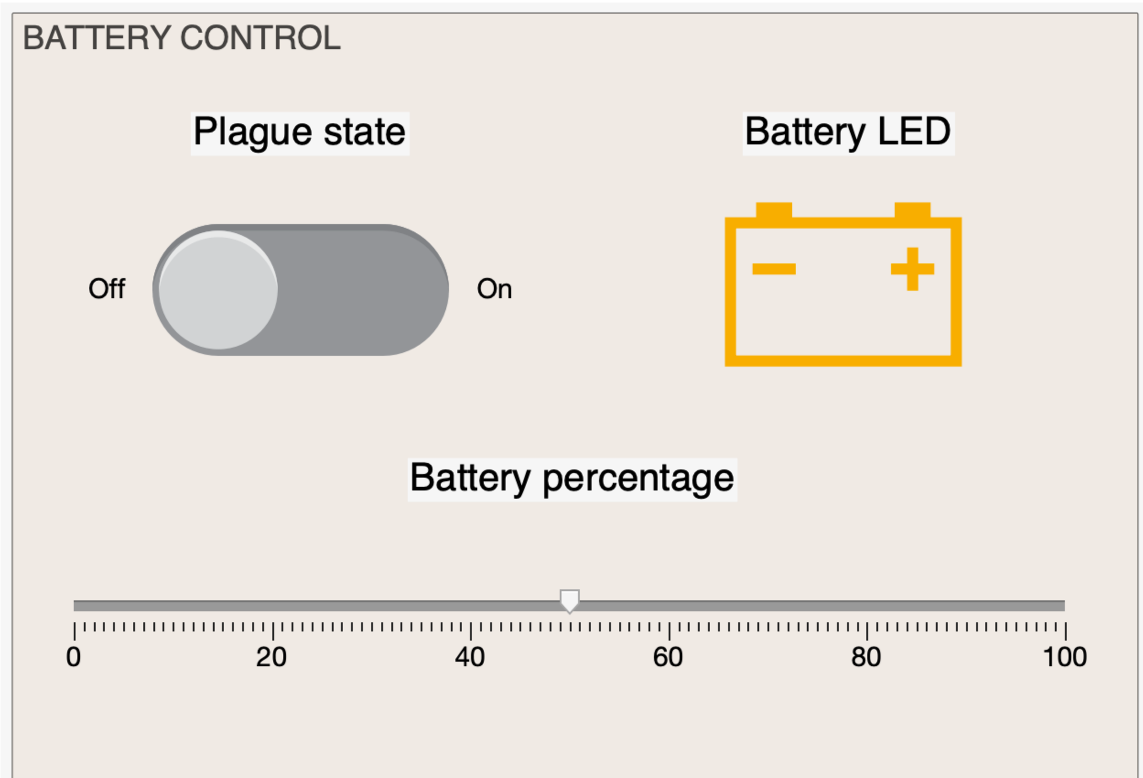


Figure 36 - BATTERY CONTROL panel

Similar to the previous panel, the BATTERY CONTROL panel comprises distinct Simulink blocks, each fulfilling a specific function in simulating the vehicle's charging process.

Plug state slider switch

The plug-state slider switch block plays an essential role in mimicking user interaction with the vehicle's charging system. This block functions as a digital two-position switch, replicating the act of plugging or unplugging the vehicle from a charging station. When activated by the user, the slider switch transmits a corresponding digital signal to the simulated system. This signal triggers the activation or deactivation of the charging function, allowing for realistic control over the vehicle's connection to the external power source.

Battery percentage slider

The battery percentage slider block serves as a valuable tool for researchers to manipulate the battery's state of charge during the simulation. This user-controlled slider allows for the adjustment of a continuous signal representing the battery's percentage charge. By modifying this value, developers can explore the system's performance and behaviour under a variety of battery charge levels.

Battery LED

The battery LED block functions as a visual indicator of the battery's state of charge. This block emulates a light-emitting diode (LED) in the dashboard of the vehicle that changes colour based on the received digital signal representing the battery percentage. The colour coding scheme employed within the simulation adheres to a widely recognised convention:

- red light means a critical battery level, corresponding to a charge level of 20% or less. This visual cue alerts the user to the urgent need for charging the vehicle;

- yellow light indicates a medium battery level, ranging from 20% to 80%. This indicates a condition where the vehicle still has a moderate amount of charge available, without concern for the driver;
- green light denotes a fully charged or almost charged battery, exceeding 80% charge level. This visual indication informs the user that the vehicle has sufficient range for operation.

The inclusion of the battery percentage slider combined with the status LED enhances the user experience within the Simulink environment by providing an intuitive and easy-to-interpret visual representation of the battery's state of charge.

The right-hand side of the dashboard is completed by the FAULT CONTROL panel.

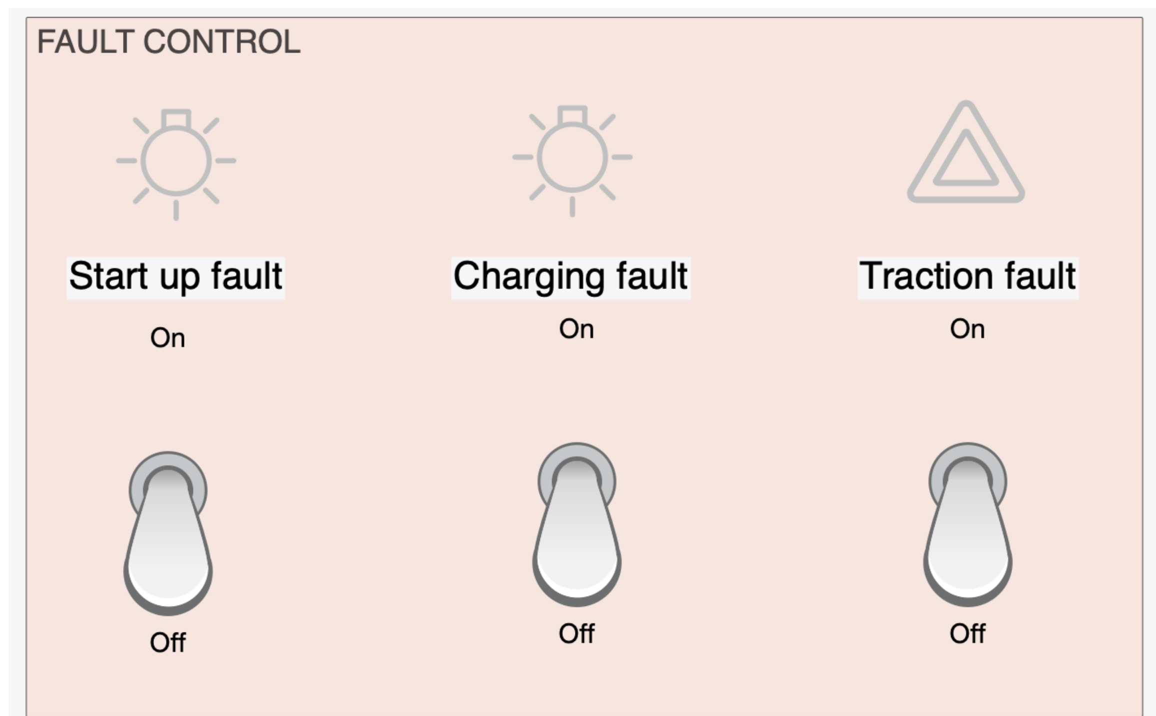


Figure 37 - FAULT CONTROL panel

The simulation incorporates a dedicated FAULT CONTROL panel, serving as a vital tool to investigate the system's behaviour under various fault conditions. This panel comprises distinct elements that facilitate the controlled introduction of faults and provide visual feedback on their occurrence. Here's a detailed exploration of these elements:

Fault toggle switches

The Fault Control Panel features three prominent toggle switches, each designed to correspond to a specific potential fault scenario. These switches, acting as digital two-position switches, empower the simulation user with the ability to activate fault injection. When a user toggles a switch to the "on" position, the corresponding fault signal is injected into the state machine. This simulated fault triggers the state machine to transition from a nominal operational state to the designated fault state associated with the activated switch. For instance, activating the "Start up fault" switch would cause the state machine to simulate a starting system malfunction.

Fault lamps

Accompanying each toggle switch is a dedicated fault indicator lamp. These lamps function as crucial visual aids, providing immediate feedback on the current fault state within the simulation. Upon activation of a specific fault through the corresponding toggle switch, the associated lamp illuminates, signifying the presence of the simulated fault. Importantly, the system employs a distinctive design for the traction fault lamp. This lamp takes the form of a red hazard triangle, visually emphasising the critical nature of traction failures, that typically occur while the vehicle is moving, pose the greatest safety risk and require immediate attention.

By combining the functionality of toggle switches and fault indicator lamps, the FAULT CONTROL panel grants users the ability to meticulously introduce and monitor simulated faults within the system. This capability facilitates a rigorous evaluation of the system's response to various fault conditions, aiding in the development of robust fault detection and isolation mechanisms for real-world vehicle operation.

Upon careful examination of the panels responsible for user interaction and fault injection, our focus now turns to the right-hand side of the Simulink project. This section houses three panels dedicated to the management and visualization of data within the simulation. The functionality and importance of these panels are here explained.

Starting at the bottom is the OUTPUT CONTROL panel.

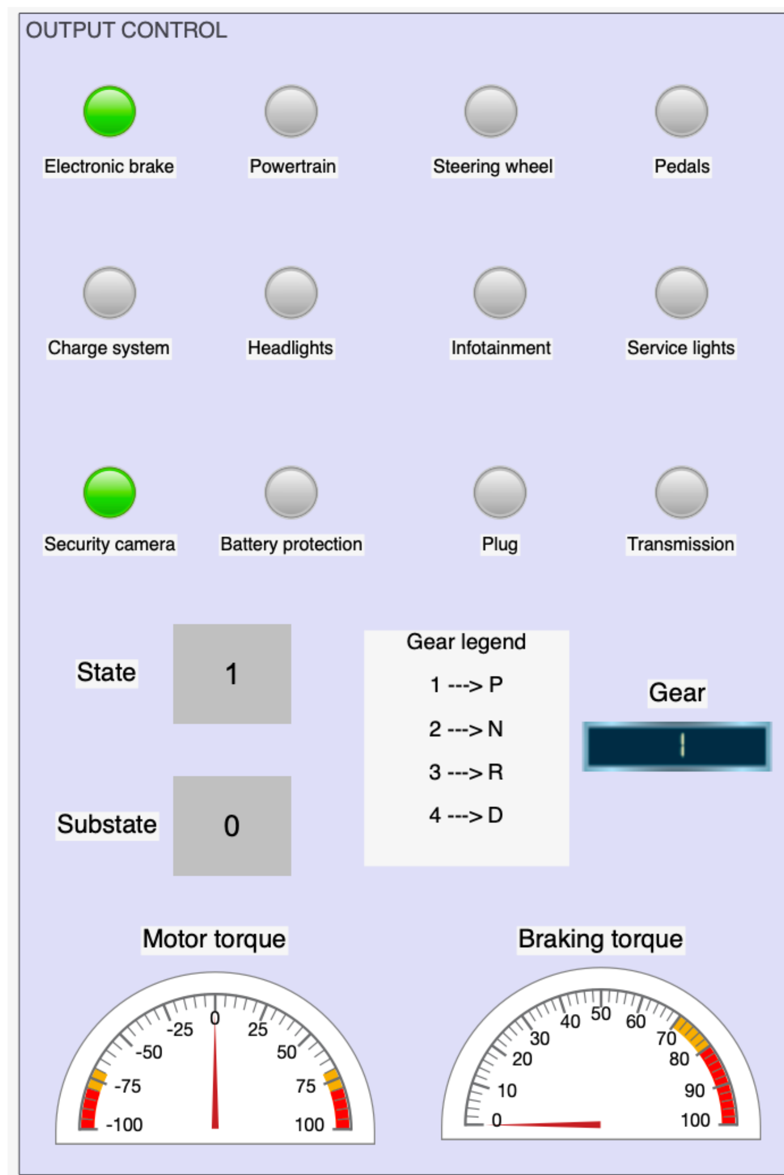


Figure 38 - OUTPUT CONTROL panel

The OUTPUT CONTROL panel is the primary means of visualising and understanding system behaviour within the Simulink environment. This panel plays a pivotal role in providing a clear and centralised view of all outputs generated by the system during the simulation run. This centralised display provides a quick and easy overview of the system throughout the simulation, enabling researchers to analyse system behaviour and gain valuable insight into system performance. Users can also easily observe the evolution of key variables associated with individual system components. By including all relevant outputs in a single location, the panel eliminates the need to navigate through different sections of the simulation model, improving the user experience and streamlining data analysis. The OUTPUT CONTROL panel takes a multi-faceted approach to data presentation, accommodating different user preferences and facilitating a more intuitive understanding of the state of the system. Here's a breakdown of the visualisation blocks used:

Status LEDs

These LEDs are directly associated with all boolean variables within the system. Their illuminated state provides a clear and immediate visual indication of the current value (active or inactive) associated with each variable. This rapid visual update enhances the user's ability to monitor the system's dynamic behaviour.

Dedicated displays

Specific displays are integrated into the panel to show key system parameters. These displays include:

- active status and sub-status: these two displays use numbers to show the current operating state and sub-state of the system. This visualisation provides a concise understanding of the overall and granular operating mode of the system;
- current gear: a dedicated display, graphically distinct from the others, shows the currently selected gear. This visualisation provides a clear and easy to interpret indication of the vehicle's transmission status.

Half gauges

The panel incorporates two half gauges to visually represent the magnitude of both the engine torque and the braking torque generated by the vehicle. These gauges provide a continuous and intuitive representation of the system's power output and braking force respectively. In particular, the gauges use a colour-coded scale to highlight critical values. The scale changes from white to yellow and red as the torque approaches its limit, providing a visual warning of potential operating limits and allowing proactive action to be taken if necessary.

This combination of visual elements within the OUTPUT CONTROL panel provides a comprehensive and user-friendly interface for examining system output. Multimodal data presentation allows for a more efficient and refined understanding of system behaviour during simulation runs.

Completing the dashboard of the Simulink environment are two panels dedicated to the detailed management of the variables used by the overall project. These panels, named the INPUT panel and the OUTPUT panel, serve as the interface for configuring the data flow within the simulation. What follows is an in-depth look at their characteristics and significance:

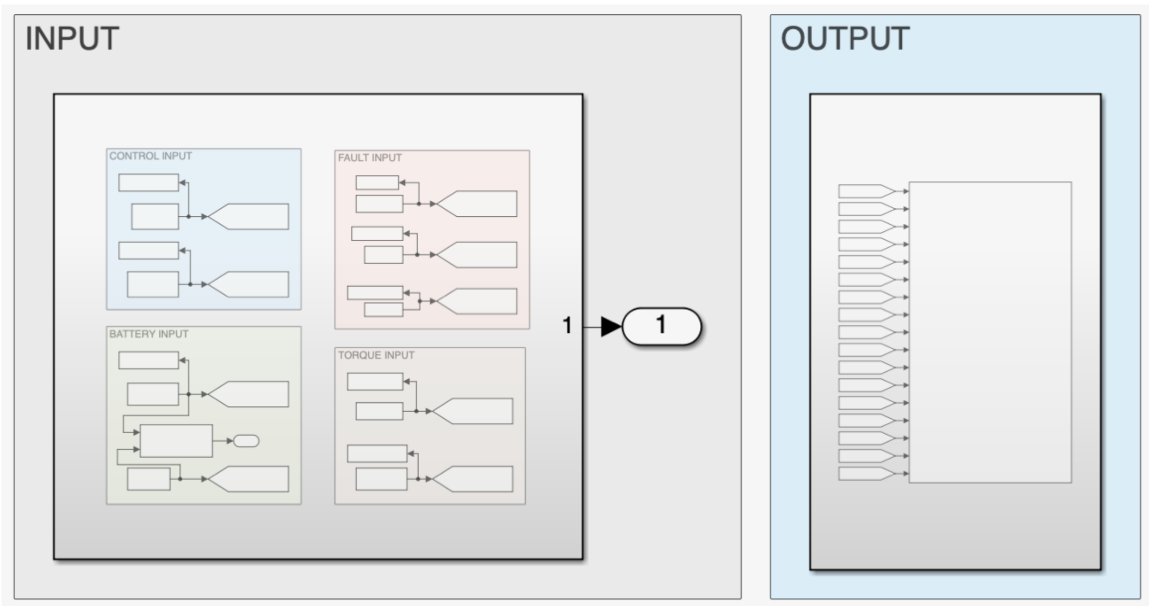


Figure 39 - INPUT and OUTPUT panels

The INPUT panel acts as the central hub for collecting and managing all the input variables used by the state machine and, by extension, the entire project. This panel contains the complete set of input variables, each accompanied by its corresponding tag block and, in some cases, function blocks, providing better organisation and enabling specific actions to be taken.

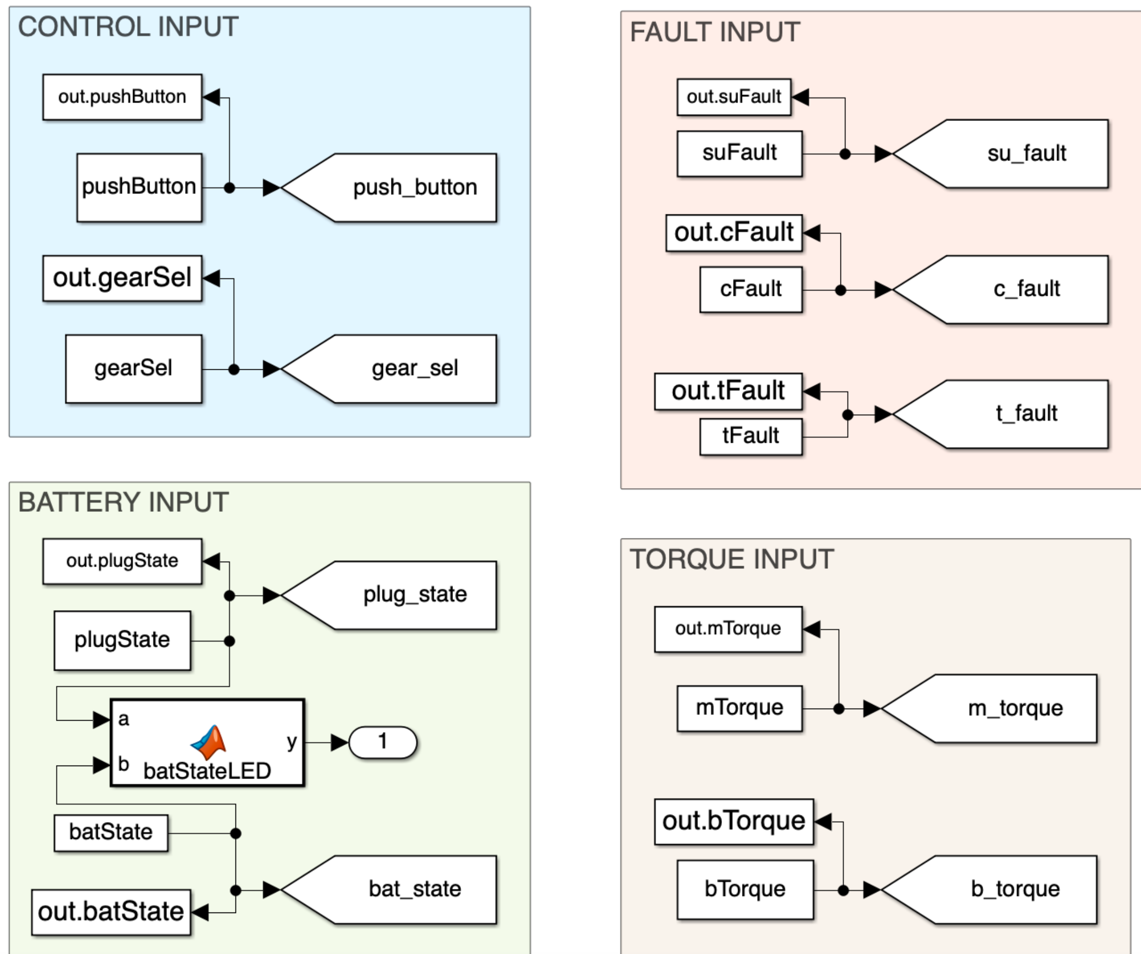


Figure 40 - INPUT panel insight

Constant and Goto blocks

The panel contains Goto blocks, which are essential for identifying and organising each input variable stored in the Constant block. A Goto block in Simulink is a signal routing tool that allows you to jump between different parts of your model without using physical connections; in fact, it works in conjunction with another block called From: the Goto block passes its input directly to the corresponding From blocks, providing a simpler and cleaner interface. This clear labelling and signal routing facilitates efficient navigation within the simulation model and optimises the process of understanding the data flow.

Function block

Beyond simple data entry, certain input variables may require additional processing or manipulation before being used by the state machine. The input panel contains a function block to deal with these cases. These blocks allow the implementation of mathematical operations, logical expressions, or other computations on the input variables, tailoring their behaviour to meet specific system requirements. In this scenario, a function is used to correctly model the behaviour of the battery LED.

In addition, the INPUT panel is further divided into sub-panels to improve organisation and facilitate user interaction. These sub-panels are categorised based on the system elements or processes they affect and are as follows:

- CONTROL INPUT: this sub-panel includes input variables related to user control inputs, such as driver commands from the MAIN CONTROL panel;
- BATTERY INPUT: this sub-panel contains input variables related to the battery model, such as the initial state of charge and the status of the battery LED;
- TORQUE INPUT: this sub-panel contains input variables related to torque generation, such as desired motor torque and braking torque commands;
- FAULT INPUT: This sub-panel manages input variables related to fault injection scenarios, allowing researchers to introduce specific faults into the simulation.

Using a structured approach with sub-panels and clear labelling, the INPUT Panel promotes a well-organised and user-friendly environment for configuring system inputs.

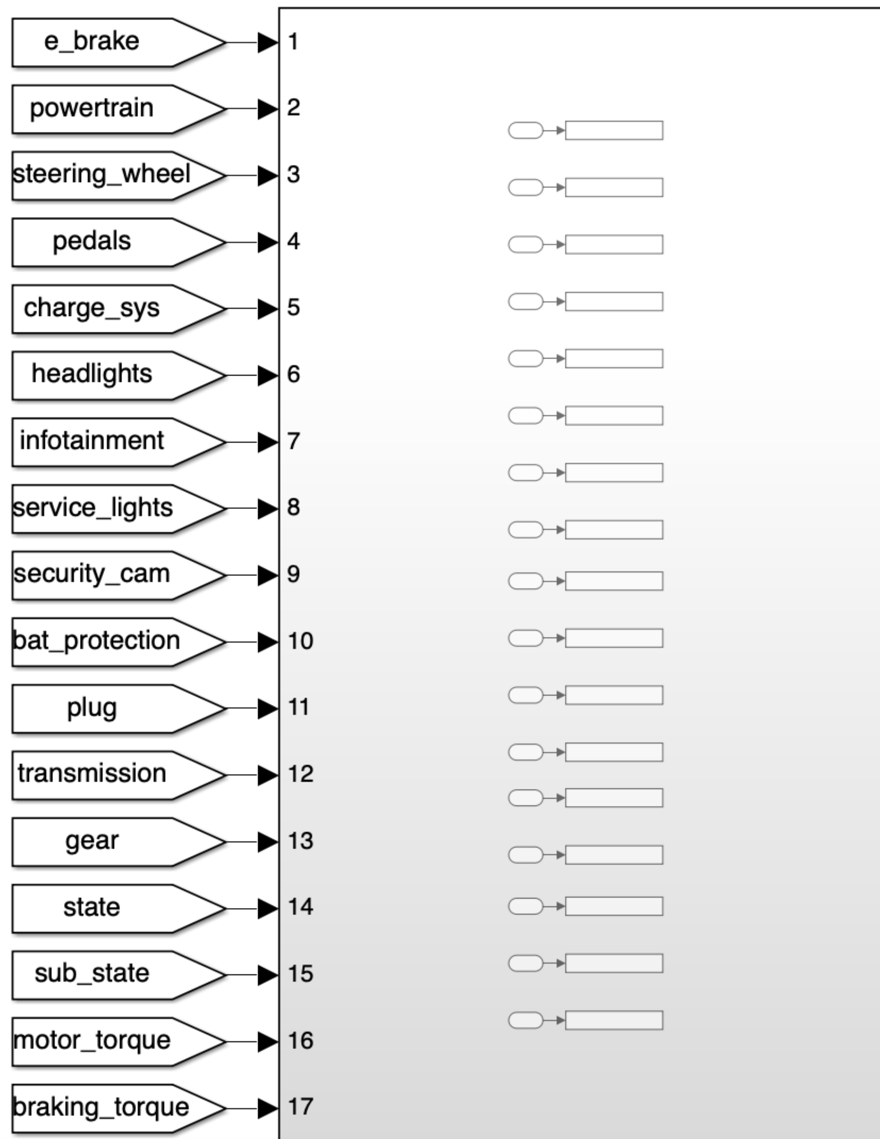


Figure 41 - OUTPUT panel insight

The OUTPUT panel serves as the central interface for managing all output variables generated by the system during simulation. These output variables are essential as they represent the system's dynamic response to various inputs and user control commands. Unlike input variables, which are defined prior to simulation execution, output variables change dynamically during the simulation run. Within this panel, you will find two main types of blocks:

From blocks

This block acts like a receiver, waiting for a signal with a specific Goto tag. When it receives a matching tag from a Goto block, it captures the input signal and outputs it. The data type of the output is the same as the input of the Goto block. As mentioned before, the From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.

To Workspace block

This block in Simulink acts as a data exporter, capturing data connected to its input port and storing it in the MATLAB workspace. During simulation, the logged data flows into the Simulation Data Inspector. When the simulation pauses or stops, the logged data is written to the workspace. This allows the data to be analysed, visualised, or further processed after the simulation run.

Examining both the INPUT and OUTPUT panels provides a full understanding of the data management structure within the Simulink environment. This structure facilitates the configuration of system inputs and the visualisation of system outputs.

After a meticulous examination of the complex model structure and associated dashboard, a decisive stage in the project is reached. This is where the groundwork is laid for the initiation of simulations and the subsequent execution of rigorous testing. This phase is critical to the overall development process. The constructed model and its interactive dashboard provide a robust platform for systematically validating the system requirements established for this project. By running a series of well-defined simulation scenarios, engineers can assess how the modelled system behaves under various conditions and user interactions. This rigorous testing process allows a detailed comparison between the expected system behaviour, as outlined in the requirements, and the actual performance observed in the simulation environment. Any deviations from the expected behaviour can be easily identified and addressed, ensuring that the final system meets its intended functionality.

The first step is to identify all the necessary tests that need to be performed to verify one or more requirements at a time. This is an extremely important part of the process as it helps to ensure that all requirements are tested and satisfied before moving on to the next stage of the V-cycle. A comprehensive test suite has been carefully planned, comprising a total of 12 verification and validation tests. These tests target specific project requirements and ensure thorough coverage of critical system functionality. Each Verification and Validation (V&V) test is performed completely within the Simulink environment. This approach is twofold:

- real-time interaction: the user interacts with the dashboard controls during simulation runs allowing dynamic adjustment of simulation parameters and facilitating exploration of the different operational scenarios. By manipulating quantities and parameters of interest within the model, it is possible to evaluate the response of the system under different conditions and check the requirements;
- data acquisition and analysis: after each simulation run, all simulation output is meticulously stored. This comprehensive data collection serves as the basis for subsequent analysis. Specialised MATLAB scripts (see appendix) are then used to retrieve and plot the relevant data and variables under investigation. This data visualisation allows a rigorous examination of the system under test conditions.

The successful execution of the simulation, coupled with the correct collection of data sets and analysis process, provides a solid platform for validating the identified project requirements. By comparing the observed system behaviour within the simulation with the expectations outlined in the requirements, any discrepancies can be easily spotted. In addition, the ability to repeat the simulation as many times as required increases the overall reliability of the V&V process. For a faster and more efficient start of the simulations, all variables are pre-allocated and some of them are already initialised within a MATLAB script inserted into the model as an initialisation function (InitFcn), i.e. a type of callback that is executed or evaluated at the beginning of model compilation.

The first is the only static test where the simulation is simply started and the model is set to OFF by default, allowing requirements related to this state to be verified without any user intervention on the dashboard. TEST 1 deals with the checking of C1 to C10 requirements:

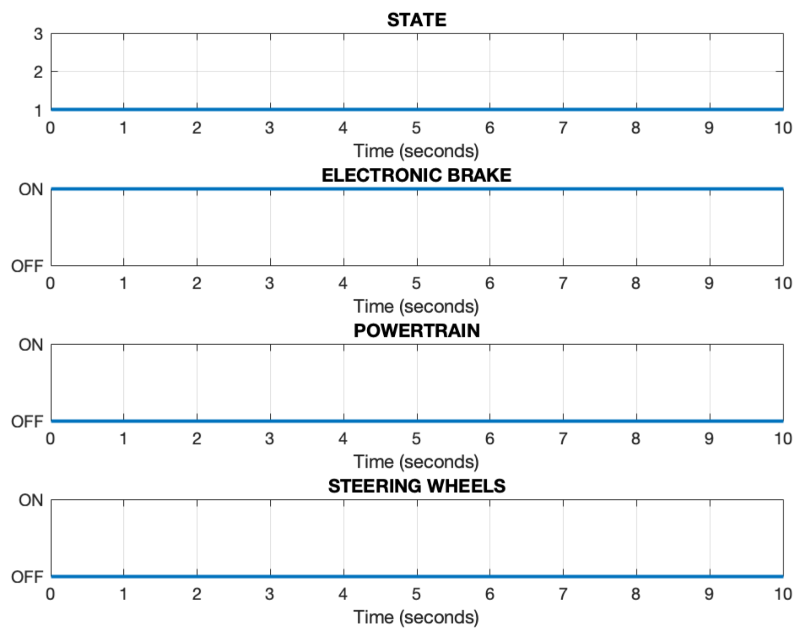


Figure 42 - Test 1_1 results

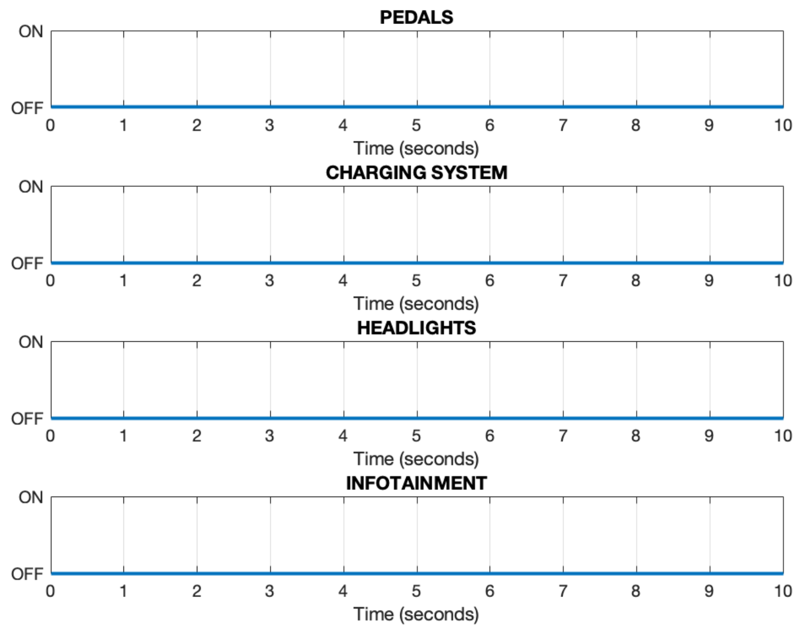


Figure 43 - Test 1_2 results

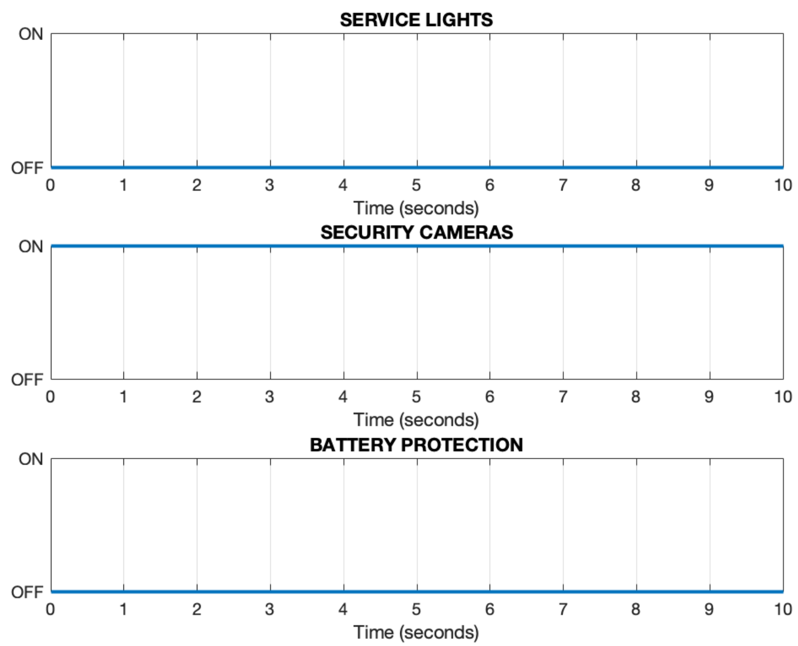


Figure 44 - Test 1_3 results

From the second test onwards, specific user actions on the control panels are required during the execution of the simulation to fully and correctly verify the requirements under consideration. TEST 2 deals with the checking of C11 to C20 requirements. These conditions are related to the CHARGING state of the machine, so the tests are designed as follows: starting from the default OFF state, the plug is connected (through the specific command on the dashboard), allowing the proper transition of the state machine to the CHARGING state; when there, the outputs are collected and analysed for their conformity with the requirements.

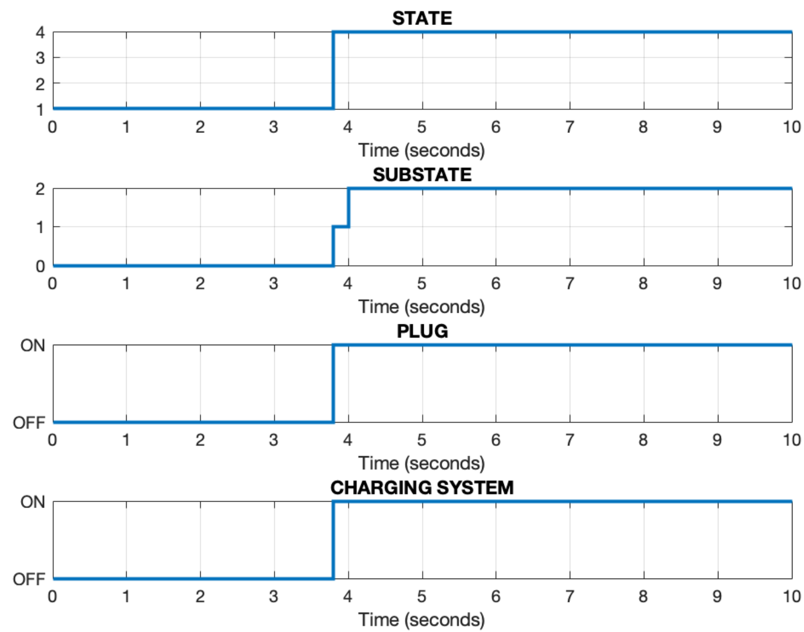


Figure 45 - Test 2_1 results

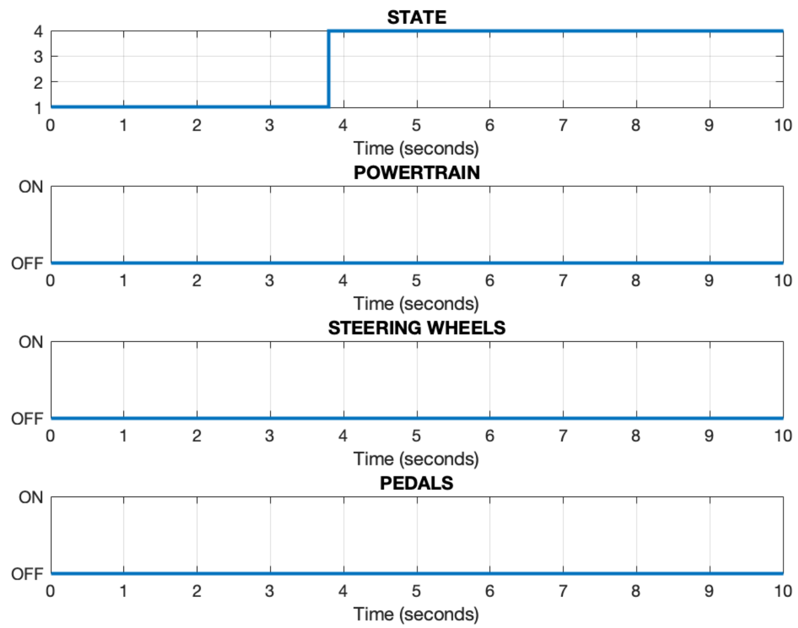


Figure 46 - Test 2_2 results

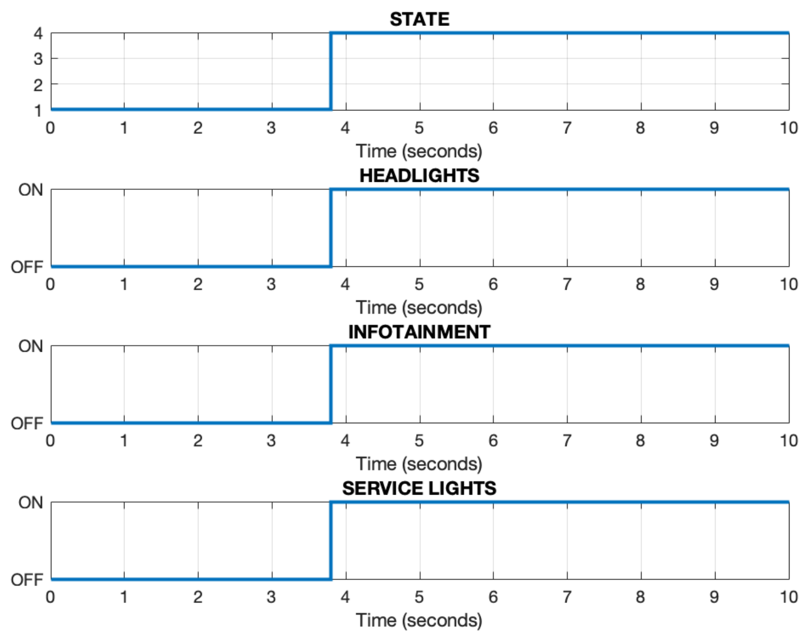


Figure 47 - Test 2_3 results

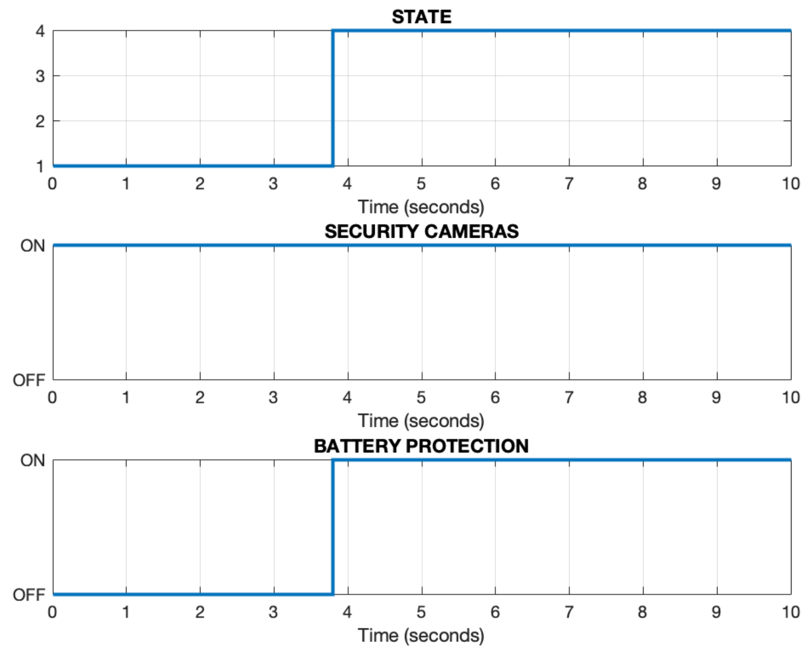


Figure 48 - Test 2_4 results

TEST 3 deals with the checking of the C21 requirement. This indicates the end of the charging process when the vehicle is plugged and has reached 100% battery charge:

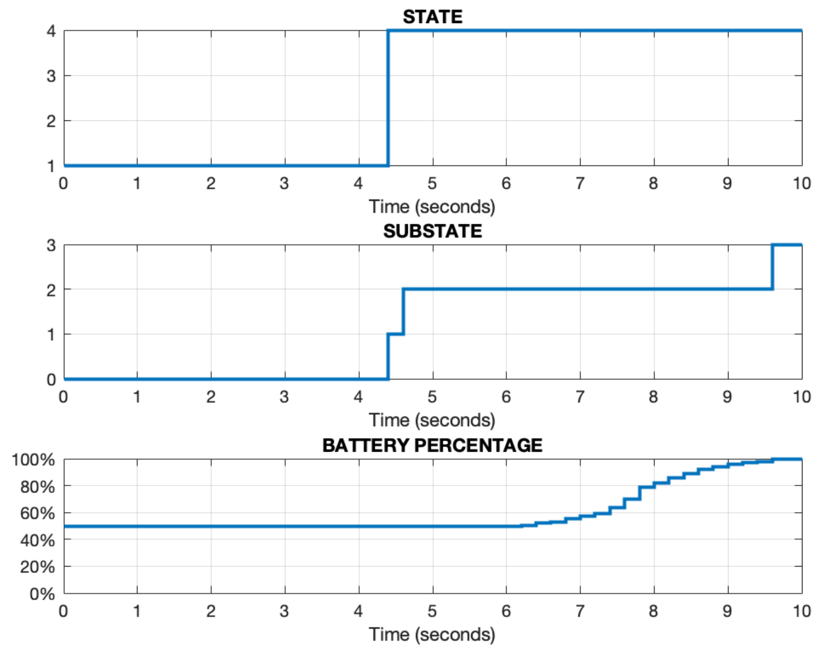


Figure 49 - Test 3 results

TEST 4 deals with the checking of C22 and C23 requirements. The fault injection technique is used to verify these requirements. When the machine reaches the desired state (in this case, the CHARGING state), a fault is simulated, acting on the appropriate toggle, to verify the correctness of the system's response to it and the next phase after its resolution:

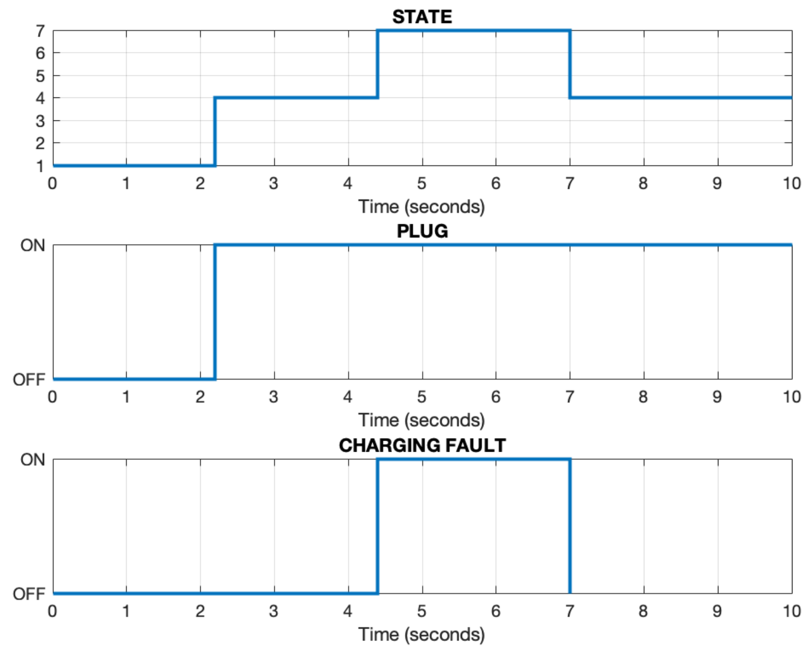


Figure 50 – Test 4 results

TEST 5 deals with the checking of C24 to C30 requirements. This set of requirements deals with the state immediately before the driving phase. During the simulation, always starting with the vehicle switched off, the vehicle reaches this state by pressing the START/STOP button, after which it is verified that all components involved in the start-up sequence proceed to their correct activation.

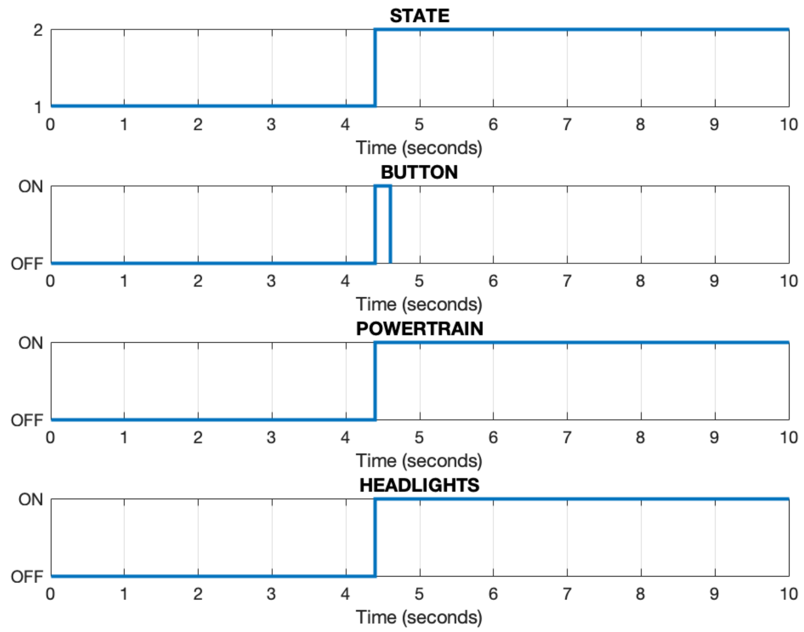


Figure 51 – Test 5_1 results

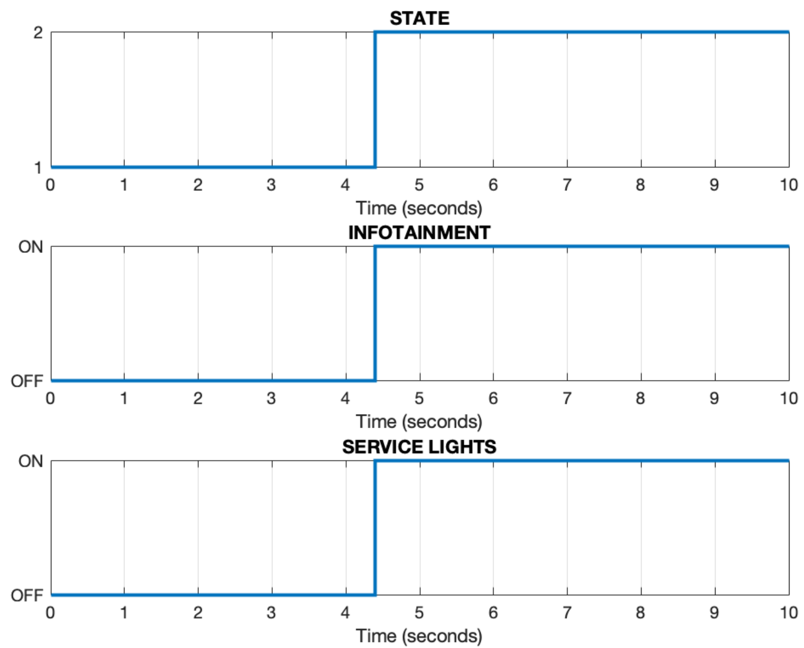


Figure 52 - Test 5_2 results

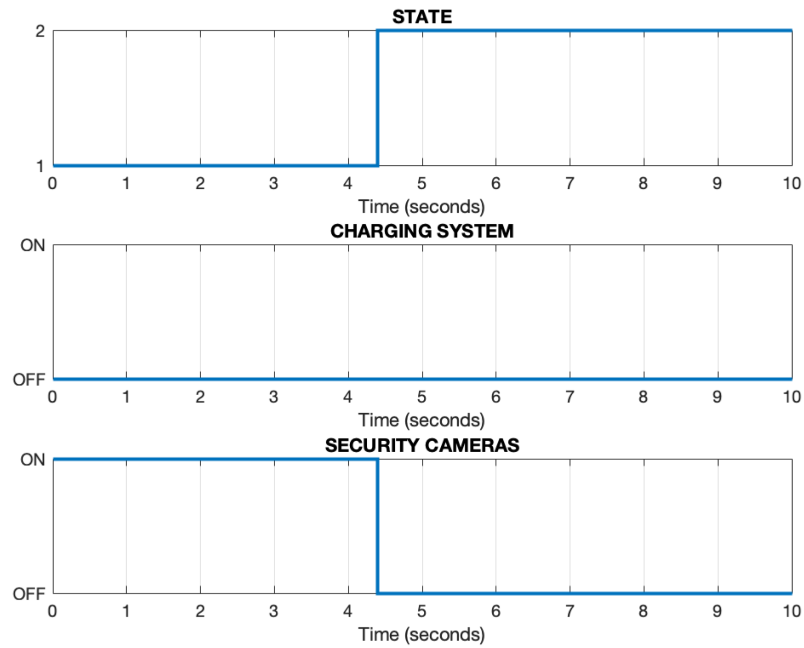


Figure 53 - Test 5_3 results

TEST 6 deals with the checking of C31 and C32 requirements. Similar to TEST 4, there is another fault injection, this time affecting the START UP state. When the system detects the fault, it immediately enters the START UP FAULT state; when the problem is fixed, the system returns to the state it was in when the problem occurred.

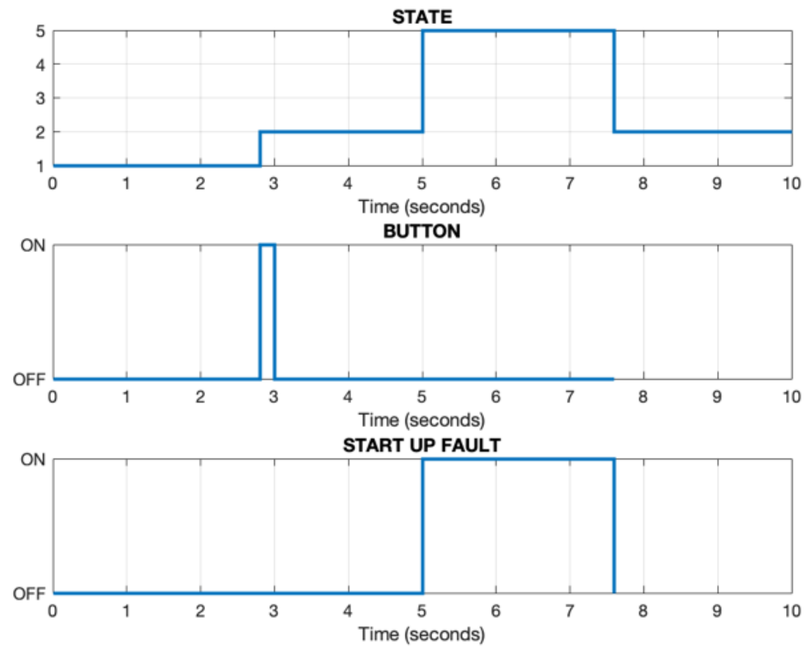


Figure 54 - Test 6 results

TEST 7 deals with the checking of C33 to C41 requirements. Verification of this set of requirements allows evaluation of the correct activation and deactivation of specific system components when entering the TRACTION state. To reach this state, it is necessary to first enter the START state and then press the START/STOP button one more time.

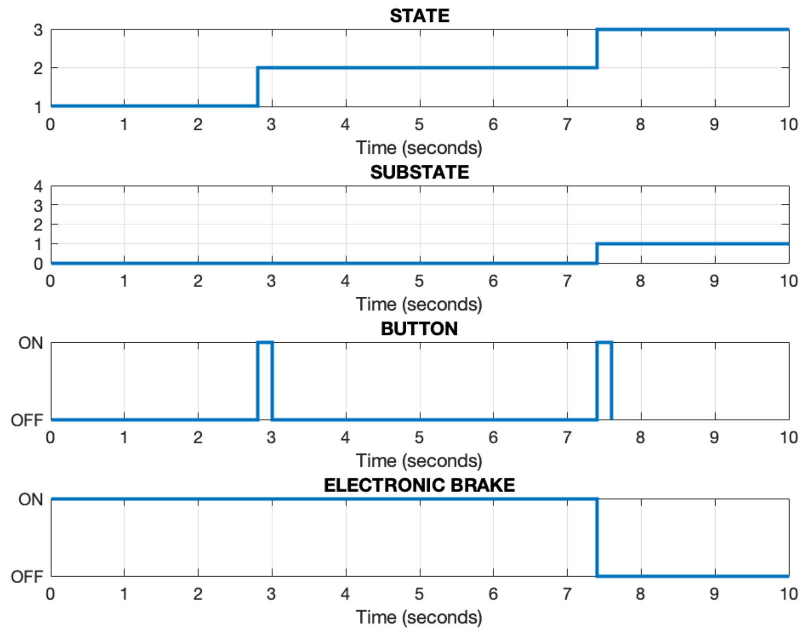


Figure 55 - Test 7_1 results

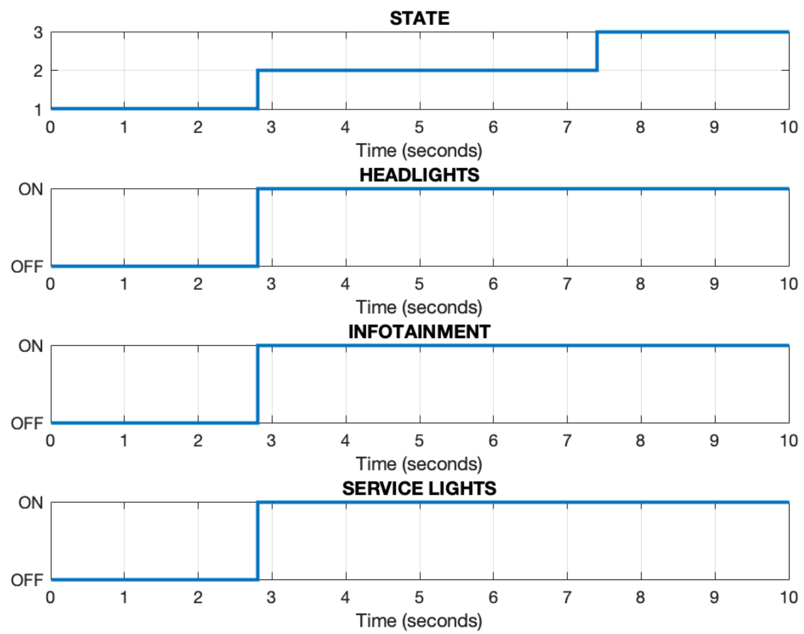


Figure 56 - Test 7_2 results

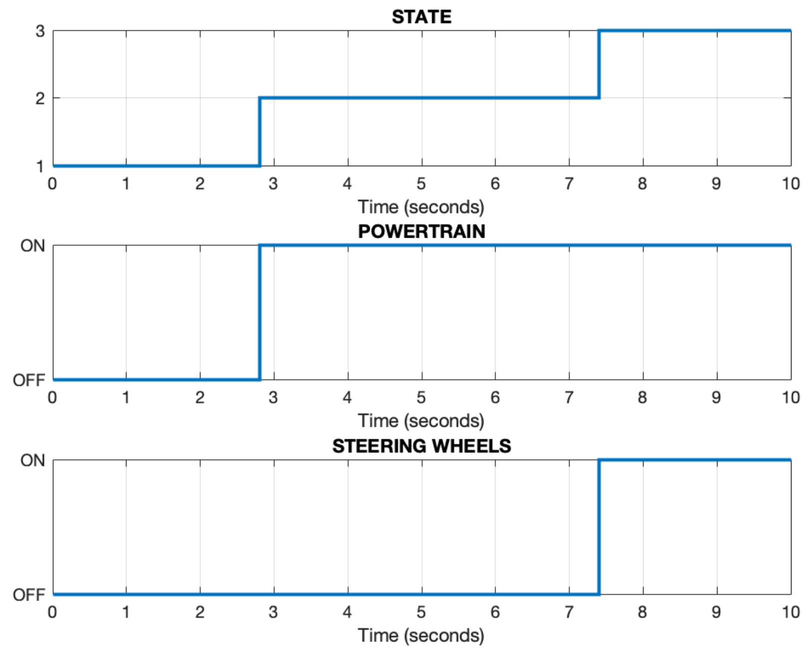


Figure 57 - Test 7_3 results

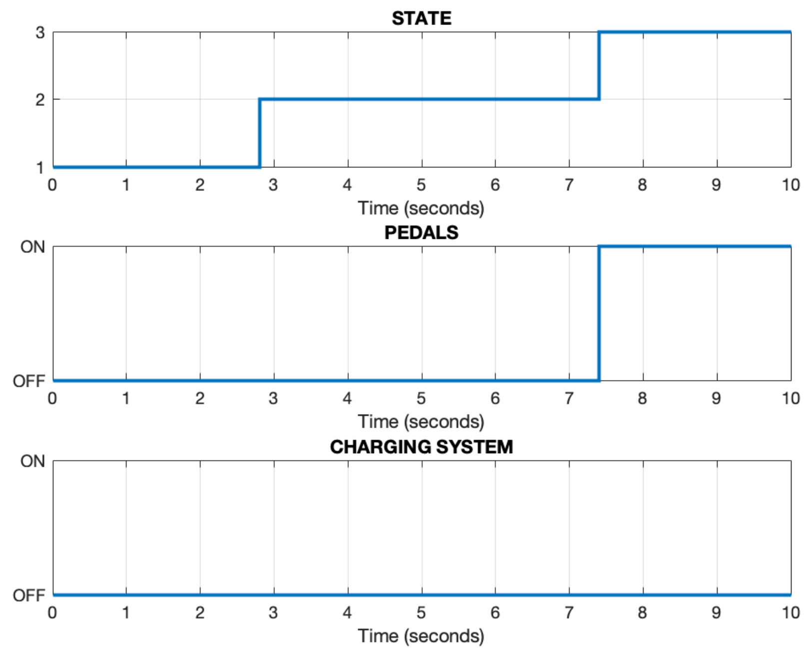


Figure 58 - Test 7_4 results

TEST 8 deals with the verification of requirements C42 and C43. In particular, it deals with the behaviour of the vehicle when the D gear is engaged. Note that the motor torque is positive, reflecting the fact that the vehicle is moving forward.

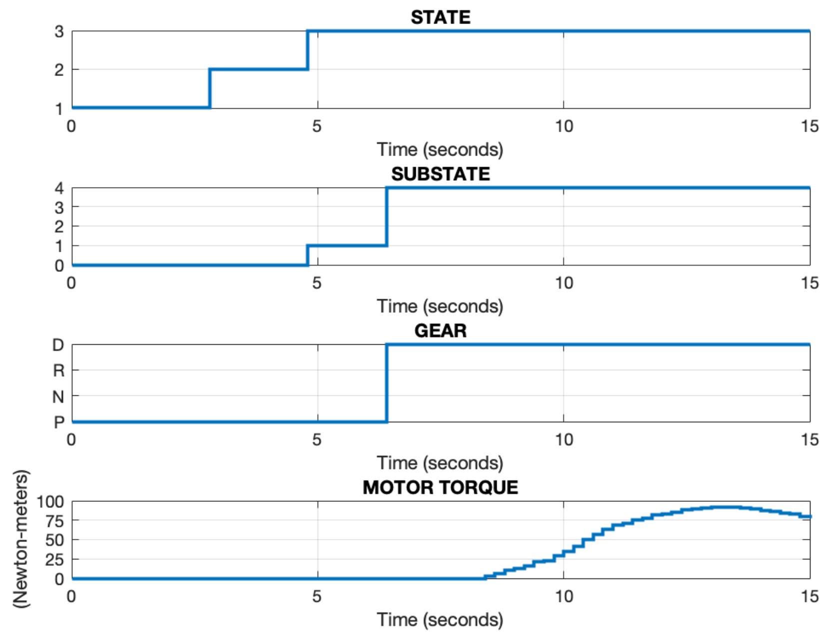


Figure 59 - Test 8 results

TEST 9 deals with the verification of requirements C44 and C45, checking the vehicle's behaviour when R gear is engaged. Note that the motor torque is negative, reflecting the fact that the vehicle is moving backward.

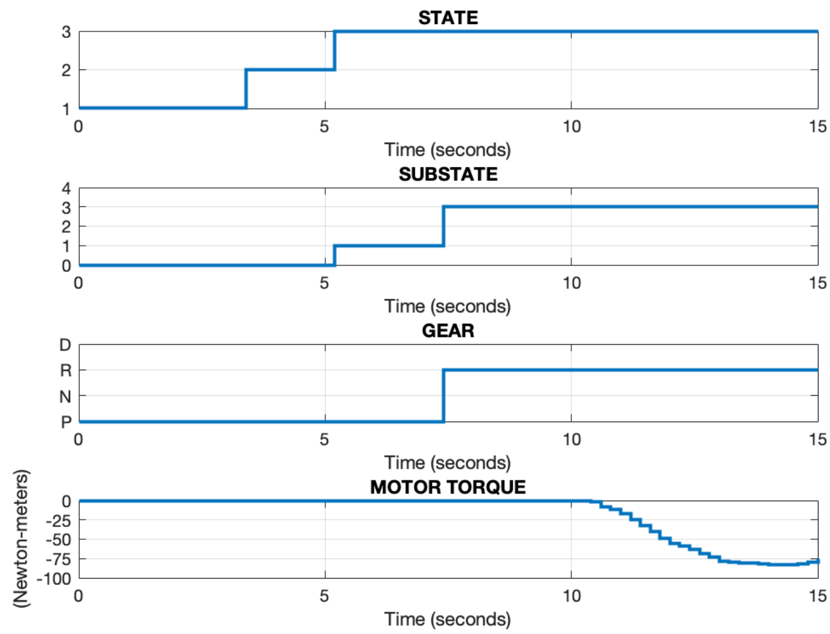


Figure 60 - Test 9 results

TEST 10 deals with the checking of the C46 requirement, checking the vehicle's behaviour when N gear is engaged. According to project requirement C46, engaging the neutral gear signifies a disengaged powertrain state. This condition is correctly indicated by zero motor torque values during the entire simulation.

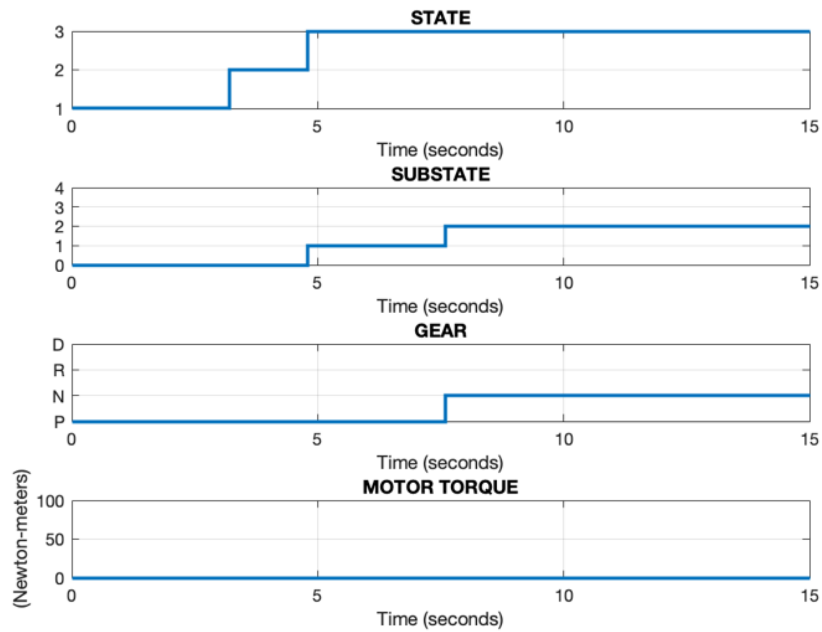


Figure 61 - Test 10 results

TEST 11 deals with the checking of C47 and C48 requirements, checking the vehicle's behaviour when P gear is engaged. To meet these requirements, it is necessary to check that the powertrain is disconnected and that it is possible to shut down the car by pressing the START/STOP button when the vehicle is in driving mode, but the parking gear is engaged.

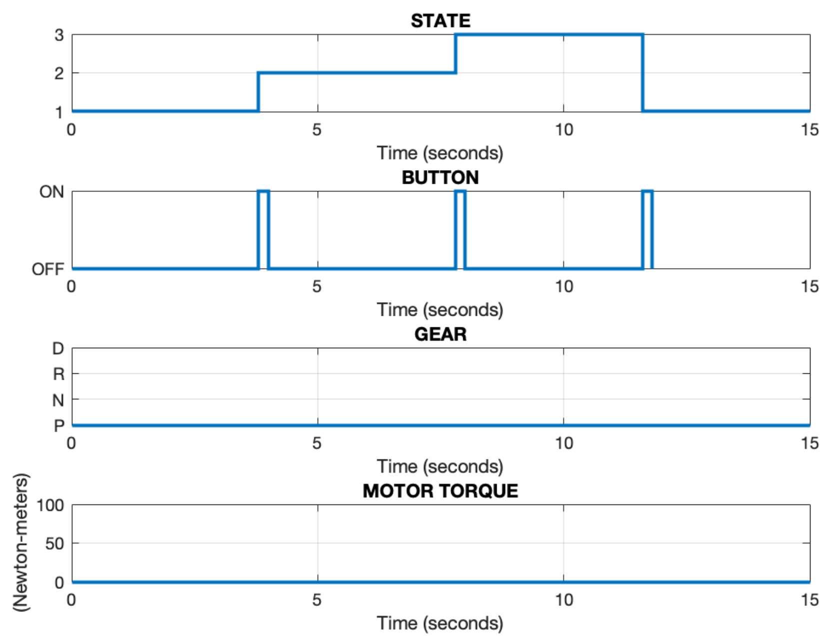


Figure 62 - Test 11 results

TEST 12 deals with the checking of C49 and C50 requirements. As seen in TESTs 4 and 6, the fault injection technique is used here to verify that the system reaches the TRACTION FAULT state and how it reacts after the fault has been resolved.

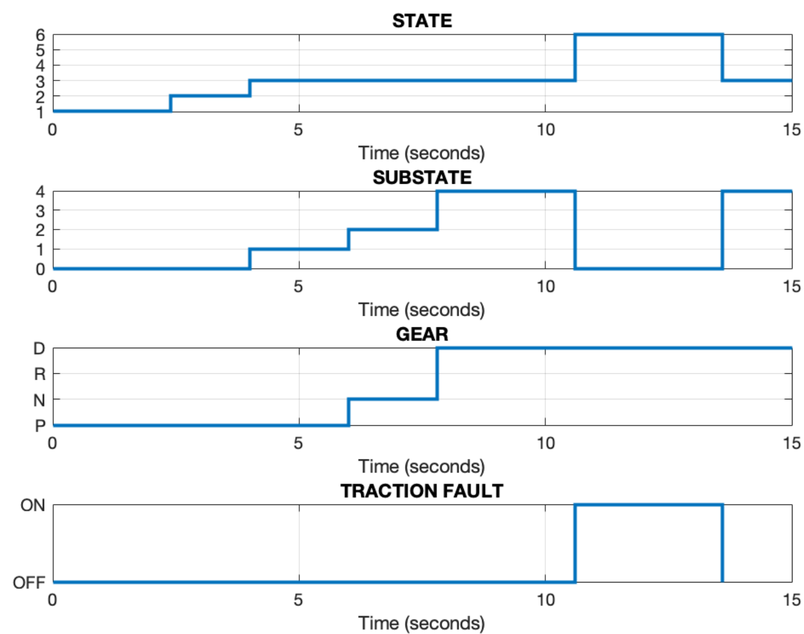


Figure 63 - Test 12 results

Extensive data analysis, facilitated by the extracted simulation results and visualised data sets, has played a critical role in demonstrating the model's compliance with all the 50 established requirements. This data serves as concrete evidence and provides a clear and objective assessment of the system response. Through this rigorous V&V process, developers gain a high degree of confidence that the simulated system will operate as expected. Successful validation of project requirements provides compelling evidence that the system design is consistent with its intended functionality. In summary, this V&V process, carefully executed within the Simulink environment, serves as a critical gateway to project success. By systematically evaluating the behaviour of the system against the established requirements, it is possible to ensure that the designed model is well-positioned for subsequent real-world implementation and fulfils the overall project objectives.

Conclusion

In conclusion, this thesis has presented a powerful demonstration of the effectiveness of Model-Based Software Design (MBSD) techniques within the development of the EVERGRIN project's Vehicle Management Unit (VMU). By focusing on the left side of the V-cycle development process, this research specifically explored the benefits of Model-In-the-Loop (MIL) testing within the system design phase. Leveraging the Simulink environment, combined with the Stateflow tool, the development process benefited significantly from the ability to create a system model that could be controlled in an interactive, real-time simulation via a custom dashboard.

This approach facilitated a robust trial-and-error process, allowing multiple considerations and optimisations from the initial generation of system states to the final stages of software testing. In fact, the iterative nature of the MBSD approach, starting with the definition of system requirements in accordance with current international standards, through the creation of the state machine, to the consequent translation by software, promoted an effective and dynamic environment for verification and refinement. This not only ensured the intended functionality and behaviour of the system under study but also allowed for continuous improvement, maximising its efficiency and adaptability.

The focus of this research has been on software development, and the natural progression from this point would involve the generation of code and its subsequent progressive integration with hardware. This procedure would indeed follow the

established V-cycle development model, specifically dealing with the lower and right sides of the cycle. After successful Verification and Validation (V&V) of the embedded code, the project will be ready to move forward with combined hardware and software during the integration phase, once again proving the power of the MBSD method.

Ultimately, the research conducted in this thesis has served a valuable academic purpose by revisiting and analysing the initial development stages for the EVERGRIN project through the V-cycle. Although the company project itself is already in the final prototyping phase, this review through an academic lens has offered significant insights into the potential of MBSD methodology for designing, developing, and enhancing complex automotive systems. The software provided will serve as a valuable reference point for future efforts to develop viable and adaptable electric vehicles, ultimately contributing to a more sustainable and efficient automotive future.

Appendix

Appendix 1 – MATLAB code for TEST 1

```
close all
clear
clc

load("test1.mat")

state = out.state;
eBrake = out.eBrake;
powertrain = out.powertrain;
steeringWheel = out.steeringWheel;

figure('Name','TEST 1_1','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), ylim([1 3]), yticks(1:3)

subplot(4,1,2), plot(eBrake,LineWidth=2), title('ELECTRONIC BRAKE')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,3), plot(powertrain,LineWidth=2), title('POWERTRAIN')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(steeringWheel,LineWidth=2), title('STEERING
WHEELS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
pedals = out.pedals;
chargeSys = out.chargeSys;
headlights = out.headlights;
infotainment = out.infotainment;
```

```
figure('Name','TEST 1_2','NumberTitle','off')

subplot(4,1,1), plot(pedals,LineWidth=2), title('PEDALS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,2), plot(chargeSys,LineWidth=2), title('CHARGING
SYSTEM')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,3), plot(headlights,LineWidth=2), title('HEADLIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(infotainment,LineWidth=2),
title('INFOTAINMENT')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
serviceLights = out.serviceLights;
securityCam = out.securityCam;
batProtection = out.batProtection;

figure('Name','TEST 1_3','NumberTitle','off')

subplot(3,1,1), plot(serviceLights,LineWidth=2), title('SERVICE
LIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,2), plot(securityCam,LineWidth=2), title('SECURITY
CAMERAS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(batProtection,LineWidth=2), title('BATTERY
PROTECTION')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
```

Appendix 2 – MATLAB code for TEST 2

```
close all
clear
clc

load("test2.mat")

state = out.state;
subState = out.subState;
plug = out.plug;
chargeSys = out.chargeSys;

figure('Name','TEST 2_1','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel('')

subplot(4,1,3), plot(plug,LineWidth=2), title('PLUG')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(chargeSys,LineWidth=2), title('CHARGING
SYSTEM')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
powertrain = out.powertrain;
steeringWheel = out.steeringWheel;
pedals = out.pedals;

figure('Name','TEST 2_2','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(powertrain,LineWidth=2), title('POWERTRAIN')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,3), plot(steeringWheel,LineWidth=2), title('STEERING
WHEELS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
```

```
subplot(4,1,4), plot(pedals,LineWidth=2), title('PEDALS')

grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
headlights = out.headlights;
infotainment = out.infotainment;
serviceLights = out.serviceLights;

figure('Name','TEST 2_3','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(headlights,LineWidth=2), title('HEADLIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,3), plot(infotainment,LineWidth=2),
title('INFOTAINMENT')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(serviceLights,LineWidth=2), title('SERVICE
LIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
securityCam = out.securityCam;
batProtection = out.batProtection;

figure('Name','TEST 2_4','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(3,1,2), plot(securityCam,LineWidth=2), title('SECURITY
CAMERAS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(batProtection,LineWidth=2), title('BATTERY
PROTECTION')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
```

Appendix 3 – MATLAB code for TEST 3

```
close all
clear
clc

load("test3.mat")

state = out.state;
subState = out.subState;
batState = out.batState;

figure('Name','TEST 3','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(3,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel('')

subplot(3,1,3), plot(batState,LineWidth=2), title('BATTERY
PERCENTAGE')
grid on, ylabel(''), ylim([0 100]), yticks(0:20:100)
yticklabels({'0%','20%','40%','60%','80%','100%'})
```

Appendix 4 – MATLAB code for TEST 4

```
close all
clear
clc

load("test4.mat")

state = out.state;
plug = out.plug;
cFault = out.cFault;

figure('Name','TEST 4','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), ylim([1 7]), yticks(1:7)

subplot(3,1,2), plot(plug,LineWidth=2), title('PLUG')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(cFault,LineWidth=2), title('CHARGING FAULT')
grid on, xlim([0 10])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})
```

Appendix 5 – MATLAB code for TEST 5

```
close all
clear
clc

load("test5.mat")

state = out.state;
pushButton = out.pushButton;
powertrain = out.powertrain;
headlights = out.headlights;

figure('Name','TEST 5_1','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), yticks(1:2)

subplot(4,1,2), plot(pushButton,LineWidth=2), title('BUTTON')
grid on, xlim([0 10])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})

subplot(4,1,3), plot(powertrain,LineWidth=2), title('POWERTRAIN')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(headlights,LineWidth=2), title('HEADLIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
infotainment = out.infotainment;
serviceLights = out.serviceLights;

figure('Name','TEST 5_2','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), yticks(1:2)

subplot(3,1,2), plot(infotainment,LineWidth=2),
title('INFOTAINMENT')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(serviceLights,LineWidth=2), title('SERVICE
LIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
```

```
chargeSys = out.chargeSys;
securityCam = out.securityCam;

figure('Name','TEST 5_3','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), yticks(1:2)

subplot(3,1,2), plot(chargeSys,LineWidth=2), title('CHARGING
SYSTEM')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(securityCam,LineWidth=2), title('SECURITY
CAMERAS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
```

Appendix 6 – MATLAB code for TEST 6

```
close all
clear
clc

load("test6.mat")

state = out.state;
pushButton = out.pushButton;
suFault = out.suFault;

figure('Name','TEST 6','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), ylim([1 5]), yticks(1:5)

subplot(3,1,2), plot(pushButton,LineWidth=2), title('BUTTON')
grid on, xlim([0 10])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})

subplot(3,1,3), plot(suFault,LineWidth=2), title('START UP FAULT')
grid on, xlim([0 10])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})
```

Appendix 7 – MATLAB code for TEST 7

```
close all
clear
clc

load("test7.mat")

state = out.state;
subState = out.subState;
pushButton = out.pushButton;
eBrake = out.eBrake;

figure('Name','TEST 7_1','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel(''), ylim([0 4]), yticks(0:4)

subplot(4,1,3), plot(pushButton,LineWidth=2), title('BUTTON')
grid on, xlim([0 10])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})

subplot(4,1,4), plot(eBrake,LineWidth=2), title('ELECTRONIC BRAKE')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
headlights = out.headlights;
infotainment = out.infotainment;
serviceLights = out.serviceLights;

figure('Name','TEST 7_2','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(headlights,LineWidth=2), title('HEADLIGHTS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,3), plot(infotainment,LineWidth=2),
title('INFOTAINMENT')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(4,1,4), plot(serviceLights,LineWidth=2), title('SERVICE
LIGHTS')
```

```
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
powertrain = out.powertrain;
steeringWheel = out.steeringWheel;

figure('Name','TEST 7_3','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(3,1,2), plot(powertrain,LineWidth=2), title('POWERTRAIN')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(steeringWheel,LineWidth=2), title('STEERING
WHEELS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
%%
state = out.state;
pedals = out.pedals;
chargeSys = out.chargeSys;

figure('Name','TEST 7_4','NumberTitle','off')

subplot(3,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(3,1,2), plot(pedals,LineWidth=2), title('PEDALS')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})

subplot(3,1,3), plot(chargeSys,LineWidth=2), title('CHARGING
SYSTEM')
grid on, ylabel(''), ylim([0 1]), yticks(0:1),
yticklabels({'OFF','ON'})
```

Appendix 8 – MATLAB code for TEST 8

```
close all
clear
clc

load("test8.mat")

state = out.state;
subState = out.subState;
gear = out.gear;
motorTorque = out.motorTorque;

figure('Name','TEST 8','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel(''), ylim([0 4]), yticks(0:4)

subplot(4,1,3), plot(gear,LineWidth=2), title('GEAR')
grid on, ylabel(''), ylim([1 4]), yticklabels({'P','N','R','D'})

subplot(4,1,4), plot(motorTorque,LineWidth=2), title('MOTOR TORQUE')
grid on, ylabel('(Newton-meters)'), ylim([0 100]), yticks(0:25:100)
```

Appendix 9 – MATLAB code for TEST 9

```
close all
clear
clc

load("test9.mat")

state = out.state;
subState = out.subState;
gear = out.gear;
motorTorque = out.motorTorque;

figure('Name','TEST 9','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel(''), ylim([0 4]), yticks(0:4)

subplot(4,1,3), plot(gear,LineWidth=2), title('GEAR')
grid on, ylabel(''), ylim([1 4]), yticklabels({'P','N','R','D'})

subplot(4,1,4), plot(motorTorque,LineWidth=2), title('MOTOR TORQUE')
grid on, ylabel('(Newton-meters)'), ylim([-100 0]), yticks(-
100:25:0)
```

Appendix 10 – MATLAB code for TEST 10

```
close all
clear
clc

load("test10.mat")

state = out.state;
subState = out.subState;
gear = out.gear;
motorTorque = out.motorTorque;

figure('Name','TEST 10','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel(''), ylim([0 4]), yticks(0:4)

subplot(4,1,3), plot(gear,LineWidth=2), title('GEAR')
grid on, ylabel(''), ylim([1 4]), yticklabels({'P','N','R','D'})

subplot(4,1,4), plot(motorTorque,LineWidth=2), title('MOTOR TORQUE')
grid on, ylabel('(Newton-meters)'), ylim([0 100]), yticks(0:50:100)
```

Appendix 11 – MATLAB code for TEST 11

```
close all
clear
clc

load("test11.mat")

state = out.state;
pushButton = out.pushButton;
gear = out.gear;
motorTorque = out.motorTorque;

figure('Name','TEST 11','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel('')

subplot(4,1,2), plot(pushButton,LineWidth=2), title('BUTTON')
grid on, xlim([0 15])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})

subplot(4,1,3), plot(gear,LineWidth=2), title('GEAR')
grid on, ylabel(''), ylim([1 4]), yticklabels({'P','N','R','D'})

subplot(4,1,4), plot(motorTorque,LineWidth=2), title('MOTOR TORQUE')
grid on, ylabel('(Newton-meters)'), ylim([0 100]), yticks(0:50:100)
```

```
close all
clear
clc

load("test12.mat")

state = out.state;
subState = out.subState;
tFault = out.tFault;
gear = out.gear;

figure('Name','TEST 12','NumberTitle','off')

subplot(4,1,1), plot(state,LineWidth=2), title('STATE')
grid on, ylabel(''), ylim([1 6]), yticks(1:6)

subplot(4,1,2), plot(subState,LineWidth=2), title('SUBSTATE')
grid on, ylabel(''), ylim([0 4]), yticks(0:4)

subplot(4,1,3), plot(gear,LineWidth=2), title('GEAR')
grid on, ylabel(''), ylim([1 4]), yticklabels({'P','N','R','D'})

subplot(4,1,4), plot(tFault,LineWidth=2), title('TRACTION FAULT')
grid on, xlim([0 15])
ylabel(''), ylim([0 1]), yticks(0:1), yticklabels({'OFF','ON'})
```

Bibliography

[1] Måns Nilsson, Björn Nykvist: *Governing the electric vehicle transition – Near term interventions to support a green energy economy*, Elsevier, Stockholm, 2016

[2] Luca Bussi, Giovanni Guida: *PROGETTO EVERGRIN – Report Progettazione Logica*, Torino, 2023

[3] Massimo Violante: *Introduction to ISO26262*, Torino, 2022

[4] Pietro Scandale, Massimo Violante: *Model Based Design of Automotive Embedded System*, Torino, 2019

[5] Adedeji B. Badiru: *Systems Engineering Models – Theory, Methods, and Applications*, CRC Press, Boca Raton, 2019

[6] Artem Oppermann: *What Is the V-Model in Software Development?*, 2023, "https://builtin.com/software-engineering-perspectives/v-model"

[7] Massimo Violante: *Introduction to model-based software design*, Torino, 2022

[8] Institute of Electrical and Electronics Engineers: *ISO/IEC/IEEE 29148:2018*, *Systems and software engineering – Life cycle processes – Requirements engineering*, Switzerland, 2018

Acknowledgements

Acknowledgements.