On Board Charger algorithm design compliant with the Model Based approach

Davide Cassarino

A thesis presented for the Master degree of
Mechatronic Engineering

Politecnico
di Torino

Supervised by professor Massimo Violante
Department of Control and Computer Engineering
Politecnico di Torino
Italy

Academic year 2021/2022

Abstract

This thesis aims at modelling an On Board Charger for an electric vehicle following the Model Based Design. This method allows to address complex problems, while making use of system models throughout the design, by means of their simulations it leads to a rapid prototyping, software testing and verification. The aforementioned task saw the realization and simulation of more than one system model, based on the provided requirements by Teoresi. The latter ones in turn were designed following the customer requests to implement a Charging Process Management solution.
MATLAB and Simulink were used throughout the whole project, moreover, within each step of the Model Based Design workflow, it was made use of additional tools of the MATLAB environment. In particular the modelling phase hinged on Stateflow, whose graphical language allowed carrying out the logic of the Charger by way of a state transition diagram.
As for the final one for each realised model at least one test harness was implemented, hence a group of properly designed inputs stimulated the Device Under Test (DUT) to take down a collection of outputs. By means of Simulink Test and the creation of test suites, running these ones against the model made possible to measure the model coverage thus performing the MIL testing.
Meeting satisfactory results led up to SIL testing, that is to say running the same suites of tests against the auto-generated code of the model in order to measure the coverage of the code. Therefore the SIL and MIL testings were executed choosing three different coverage metrics, i.e. decision, condition and MCDC. Simultaneously with the testings, Simulink Check was employed to verify if the model complied with the ISO 26262 and to make the right adjustments for the purpose of being consistent with the safety standards related to an automotive application.
Once ISO 26262 requisites were achieved, linking what follows with the previous statements, MIL and SIL testings coverage were compared to illustrate the missing coverage objectives for the auto-generated code and to eventually find a strategy to fill them.
In the end the generated code was tested by exploiting its functions with the aim of programming a micro-controller where the code was loaded into.

To my loved ones
I received from you
lots of support when
down in the mouth
Always amazed by you

Contents

Chapter 1

Introduction to Model Based Design

1.1 The need of a new approach

The importance of Model Based Design is highlighted by the fact that intelligence systems, powered by trends like electrification, connectivity, autonomy and artificial intelligence, drive the technology of daily life. As these systems get smarter, the software embedded in them incorporates ever more powerful algorithms hence increasing the lines of embedded code.[5]
Taking into account a car as a system to be analysed, it's on-topic to remark the growth of electronic components overtaking the mechanical parts. Paying particular attention on the electronic control units present on board, some collected data shows that from 2005 the lines of code increased on average of five times within ten years, thus reaching around 16 million.[4]
Following this trend, nowadays the goal is to achieve autonomous vehicles. Compared to traditional ones where computers are necessary to manage every subsystem, here they are responsible for driving too, ergo it is possible to foresee that the number of code lines is outstanding and more important safety must be assured. Just to give some background information, an autonomous electric car with its advanced propulsion, perception, navigation, and safety features will run on an estimated 250 million lines of embedded code.[5]
The increase of software complexity is therefore a matter of significant relevance, moreover its rising leads the software development to become one of the dominant cost items in noteworthy sectors as automotive, avionic and industrial markets. Furthermore to stress its importance, in the automotive field, the shortening of time to market is a reality that all companies must face to be competitive. This can be explained by the shortening of automobiles life cycle.
The aforesaid aim of automotive industry is a fully connected and self-driving car, consequently cars and in general vehicles will rely more and more on electronic components. The latter ones have a life span of thirteen years but mostly tend to last for four or five years, a life cycle considerably shorter than traditional automotive. Nonetheless, for sake of completeness, the reduction of go to market timelines is not only due to the convergence of consumer electronics with automotive technologies but also to several other factors such as the technologies progresses and the need to meet consumers request[3], however they will not be discussed anymore.

What was set forth previously states more demanding product development timelines and the trend shows they will be shortened even further in the future, thereupon it is straightforward the necessity to tackle software complexity, keeping in mind that safety standards must always be respected. On account of this it is possible to cope with complexity making use of a structured process, a more efficient code development, an efficacious validation and a constructive way of reusing the code.
Model Based Design guarantees each of the illustrated points since it follows a reference workflow defined by the IEC, which stands for International Electrotechnical Commission, an organisation that together with the ISO, acronym of International Organization for Standardization, is responsible for standards definition in the electronic field and the ones related to this, while the latter takes up a broader application spectrum.
Based on the ISO 26262, for the automotive safety standards, and on the aforementioned reference workflow a systematic validation and verification process is set out for models and generated code. To go through the whole process it is made use of proper qualification and certification tools.

1.2 Reference workflow

The reference workflow in the Model Based Design approach, adopted in the design process, starts from system requirements, through which software requirements are extracted and used to define the software architecture.

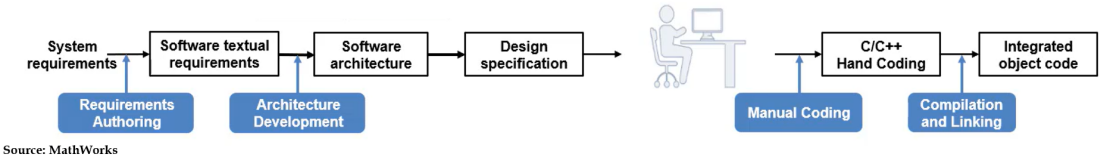


Figure 1.1: Traditional workflow

From this point on the main differences with the traditional workflow are visible.

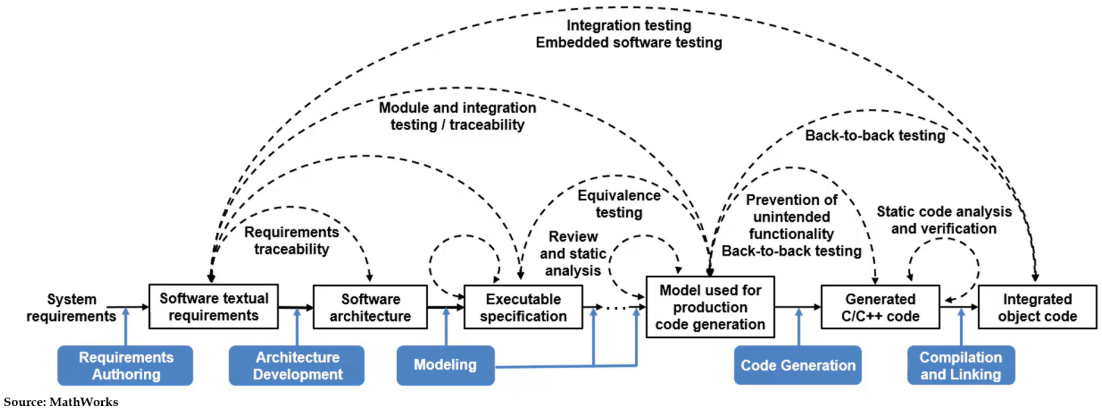


Figure 1.2: Model Based Design reference workflow

By means of suitable tools, carrying through with the system model leads to an executable specification, namely it is possible to simulate the model thereby to find and fix bugs in the course of the design process, against the traditional workflow where errors can be observed only in the final phase of the process. This huge improvement cut costs and contemporaneously saves time.
Moreover, through the inclusion of additional information, the code, that will afterwards be integrated into the target hardware, is generated automatically by a specific tool. Thanks to this step the errors made by hand coding, as in the traditional workflow, are avoided and the design timeline is shortened even more.
In conclusion, only by the usage of the reference workflow, throughout the whole process each step undergoes a continuous verification and validation in order to be compliant to the safety standards (ISO 26262 for the automotive field), requirements traceablility is guaranteed, thus linking requirements to the model and the generated code, and it eases going back to the steps of the whole workflow. What was stated above makes the Model Based Design reference workflow certifiable to develop critical embedded software and it was approved by the certification authority TÜV SÜD.

Chapter 2

Requirements

This chapter is a preamble of the system requirements acquisition and conversion into software requisites. A system overview is presented below, following the strategy for modeling the signals sent via E-CAN bus by nodes.

2.1 System overview

The system analysed in this document is an On Board Charger.
This was commissioned to Teoresi, which, based on the customer requests, decided to adopt a strategy to tackle the Charging Process Management grounded in E-CAN communication. It will be gradually described below.

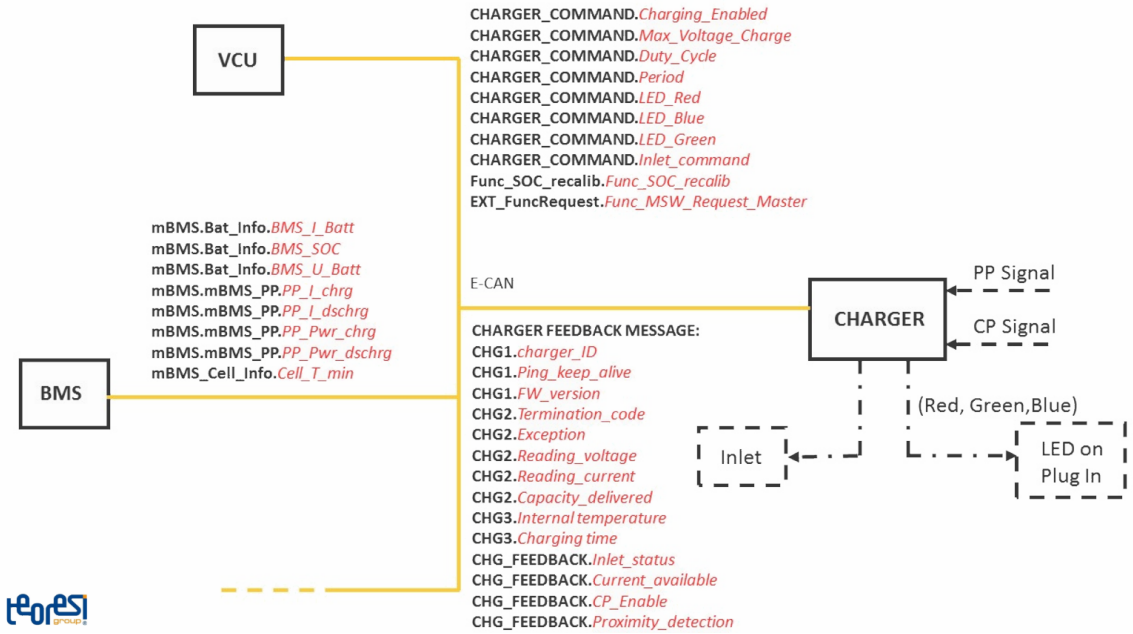


Figure 2.1: Nodes connected through the E-CAN with some of their related messages

As displayed in Figure 2.1, the nodes connected through the E-CAN are the Charger, the Vehicle Control Unit (VCU), the Battery Management System (BMS) and the Gateway (GTW), that is missing. By means of messages a node sends over a command to the E-CAN, subsequently read by the other nodes listening to the E-CAN.

The node which the command was directed to in turn replies through a feedback. Now the Charger is responsible for managing the charging process, but the communication mostly with the Vehicle Control Unit lets the process evolve. This is to say that in the design course it was important to have it clear in mind in order to bring the logic behind the Charger to fulfillment.
MATLAB and Simulink are the software employed to carry out each step of the reference workflow along with other related tools of the MathWorks environment. The latter will be described in more detail at a later stage, supported by an exhaustive analysis of the utilisation that was made of them.

2.1.1 Reference signals

The given specifications follow a division in chapters, centered on different issues, making it possible to adopt a structured path to bring to completion the final project. At first it was mandatory to address how to handle and arrange reference signals, indeed every node has its associated messages made up by various signals.
Every single signal copes with a diverse task, which is why signals are listed in tables where all the essential information are collected to properly face the design process.

Message	Cycle [ms]	ID	Signal Name	Startbit	Length [bit]	Value Description	Factor	Maximum	Byte Order
CHARGER_COMMAND	**100**	0x618	Max_Voltage_Charge	0	16		0.1		motorola
CHARGER_COMMAND	**100**	0x618	Period	16	16		0.01		motorola
CHARGER_COMMAND	**100**	0x618	Duty_Cycle	32	8		1	100%	motorola
CHARGER_COMMAND	**100**	0x618	Charging_Enabled	40	1	0x0=Charging process not enabled (current setpoint=0) 0x1= Charging process enabled(current setpoint=PP_I_chrg)	1		motorola
CHARGER_COMMAND	**100**	0x618	LED_Red	41	1	0x0 = OFF 0x1 = ON	1		motorola
CHARGER_COMMAND	**100**	0x618	LED_Blue	42	1	0x0 = OFF 0x1 = ON	1		motorola
CHARGER_COMMAND	**100**	0x618	LED_Green	43	1	0x0 = OFF 0x1 = ON	1		motorola
CHARGER_COMMAND	**100**	0x618	Inlet_command	44	2	0x0 = Not used 0x1 = Unlocked 0x2 = Locked 0x3 = Keep Last Status	1		motorola

Figure 2.2: This table contains CHARGER_COMMAND message information

The table above displays the signals details of the message CHARGER_COMMAND, sent over the E-CAN by the Vehicle Control Unit to the Charger.
Dissimilarities are present among the tables included in the documents, such as the usage of range value instead of minimum and maximum or the absence of one or both of the last two mentioned. This issue is usually due to the fact that more than one person takes care of putting down the specifications. As a matter of fact while going throughout them it occurred to find other inconsistencies. Nonetheless the presence of the fundamental characteristics led to move forward to the signals representation in the software environment.

Signals representation in Simulink

To accomplish this task it was decided to make use of the Bus Editor in Simulink, hence to get together all signals belonging to a message as elements of a Bus type. A naming convention was established and consists of the message name followed by the word *type*. Therefore the Bus type named after CHARGER_COMMAND message is CHARGER_COMMANDtype.
Based on the categories at the top of every column of the table, each element of the bus was updated following those instructions. For instance in Figure 2.2 the third element of CHARGER_COMMANDtype is called after Duty_Cycle, the length bit implies that 8 bits are mandatory to represent the signal. Furthermore in conjunction with the absence of negative values, as its maximum is 100 and its minimum implicit value is 0, the *uint8* DataType was selected.
As previously mentioned, the specifications writing was a task accomplished not only by a single person, in fact for a few amount of signals their description was ambiguous. Consequently their software representation in terms of DataType came from a designer decision.

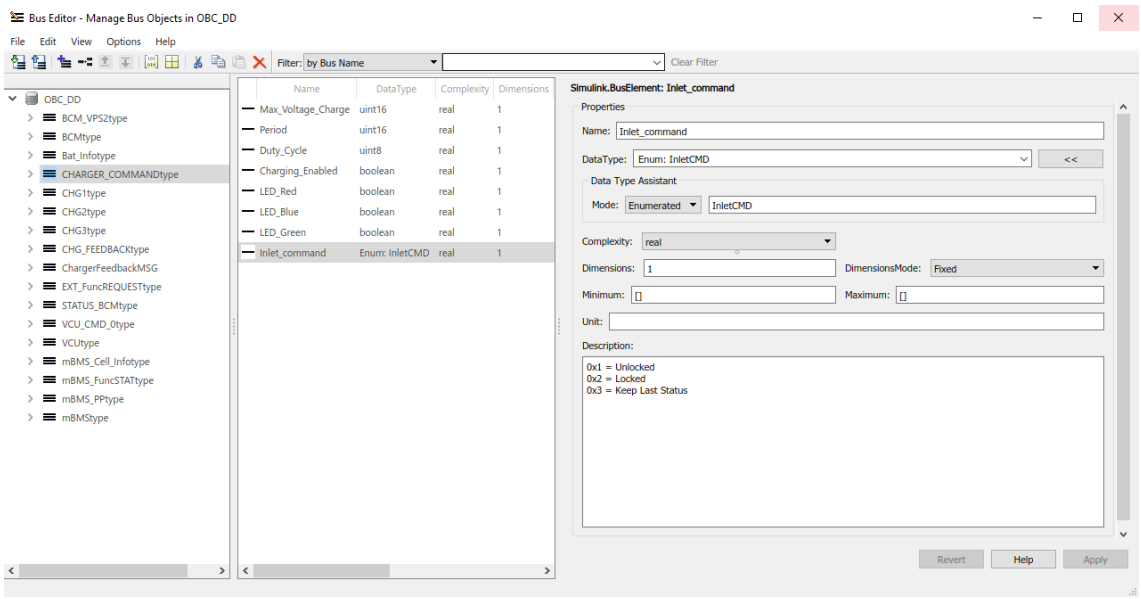


Figure 2.3: Bus Editor window from Simulink

The purpose of the other categories are out of the Simulink description, except for the value description which is inserted in the dedicated section.
Figure 2.3 shows on the left the complete list of the buses, in this way the signals are utterly extracted from the specifications and translated in Bus Objects. At the top list it is noteworthy to highlight the file path, in other words every Bus Object is stored in the data dictionary OBC_DD.
The benefits connected to data dictionary will be thoroughly discussed in the next chapters, so far it is a persistent repository. It allows to store and share data, types (an example is the Bus Object) and configuration sets between Simulink projects. This came with great usefulness, and this matter was touched upon in the previous sections, since the Charger was modeled more than once, coming out each time with an improved version of it.
Moving to the centre the set of elements of the selected bus is listed, thus the signals

belonging to the message, while on the right the details of the selected element.
Figure 2.3 shows the Inlet_command signal whose DataType is the Enumerated named InletCMD. This choice comes out with the possibility to have a clearer design and thereupon simulation of the model, since an Enumerated signal can be equal only to the words defined in its class file in MATLAB, to which an integer value is linked.

InletCMD.m

```
1  classdef InletCMD < Simulink.IntEnumType
2      %InletCMD
3      enumeration
4          Unlocked(1)
5          Locked(2)
6          KeepLastStatus(3)
7      end
8  end
```

Figure 2.4: The MATLAB file IntelCMD defines the class of the same name

For a better understanding, Figure 2.4 depicts the class definiton named InletCMD as a Simulink Integer Enumerated type. A signal whose DataType is IntelCMD assumes the integer value 1 if equal to *Unlocked*.
A bus element can also be another bus, selecting the relative bus type as DataType. To support this, CHARGER_COMMAND is grouped along with other two buses in a single bus called VCU whose DataType is VCUtype, listed in Figure 2.3.

Chapter 3

Design

First part of the specifications was meant to define the reference signals, extensively discussed in the previous chapter, while the ones that come after hinge on explaining the charging process.
What follows is indeed the implementation of an executable model. The completion of this purpose entailed the extensively use of Stateflow, essential to finely design the Charger logic and properly cope with bugs thus optimising time management. Particular care was taken of the Inlet lock management and the SOC recalibration procedure, each one having a dedicated section.

Stateflow

Stateflow is a graphical programming environment based on finite state machines. With Stateflow the user can test and debug the design, consider different simulation scenarios, and generate code from the state machine. Finite state machines are representations of dynamic systems that transition from one mode of operation (state) to another. State machines:

- Serve as a high-level starting point for a complex software design process.
- Enable to focus on the operating modes and the conditions required to pass from one mode to the next mode.
- Help to design models that remain clear and concise even as the level of model complexity increases.

Control systems design relies heavily on state machines to manage complex logic. Applications include designing aircraft, automobiles, and robotics control systems.[7]

3.1 Inlet lock management

The modeling process started focusing on the inlet lock management, hence building up a chart able to provide a graphical description of the logic required to handle this task and the charging manual stop.
A model of the Vehicle Control Unit was developed and enhanced over time in order to simplify the design of the Charger. Connecting the two charts improved the time management. It led indeed to the correct logic flow quicker and to less use of input signals for testing the model, thus creating tests harness easier to manage.
Following a top down strategy the problem was divided into three parts: start of charging, charging process and end of charging.

Start of charging

In this step, the inlet shall switch its status, from unlocked to locked. It happens when the plug is inserted in the socket. The Charger can read the plug presence, therefore once inserted it wakes up and sends feedback to the Vehicle Control Unit. The latter answers to the Charger through a command to close the inlet, which in turn instructs the plug lock actuator to lock the plug and sets a feedback signal.
The Control Unit reads the signal on the CAN and sends a request to the Battery Management System in order to close the main switches. Thereupon the BMS answers with the main switches feedback state and if closed the VCU sends the command to start the charging process to the Charger.

Charging process

During the charging process, the plug shall remain locked. The Vehicle Control Unit constantly reads the E-CAN to check feedback messages sent by the Charger.
In normal condition the Charger notifies the presence of the plug and its lock by the plug lock actuator, while the VCU renews the order to close the inlet.
In case of fault the Charger communicates the warning state through a proper message and the Control Unit gives the Charger a dedicated instruction.

End of charging due to manual stop

At the end of the charging process the plug shall remain locked until a manual unlocking occurs due to electrical doors unlocking. The user can stop the charging process using an external command.
Two new frames were added in order to manage this function. The Gateway node mirrors the Body Control Module, from the B-CAN (Body Controller Area Network) to the E-CAN, consequently two signals are used for this task. A signal to identify the electrical locking/unlocking state of the doors and another one to represent the unlock button pressing, which, together with the remote control or physical key, changes the value of the first one.

3.1.1 Manual stop procedure

To interrupt the charging process two cases are considered depending on the value of the doors lock. If the vehicle doors are unlocked, pressing the unlock button once causes the transition to the manual stop procedure in the state machines. If the vehicle doors are locked, their unlocking is necessary to impel the state machines to start the manual stop procedure.
The Vehicle Control Unit sets the procedure, shown in Figure 3.1, and drives the Charger to follow the commands in order to unlock the plug and put in standby or end the charging process.

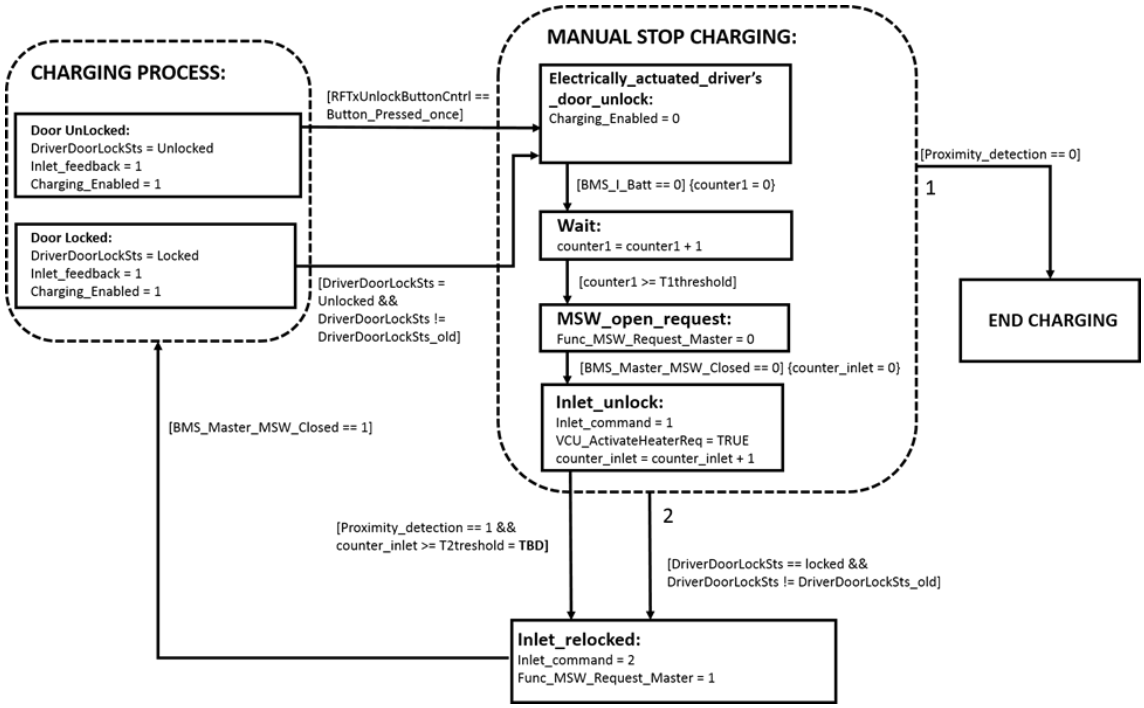


Figure 3.1: Vehicle Control Unit manual stop procedure

The VCU sends on the CAN the command to stop the charging process. The Charger reads it and shortly after sets the charging current set-point to zero. The Control Unit awaits the current to reach zero checking for its value, periodically updated by the Battery Management System. Subsequently it waits for a set threshold before sending the request to the BMS to open its main switches. The latter answers with positive feedback thereupon the VCU commands the Gateway to discharge the capacitors and the Charger to unlock the plug.
The Charger is responsible for the detection of the plug, whose status is updated through the CAN by a feedback signal. The VCU answers to the Charger issuing the order to transit to the idle state if the plug is removed. If not, it broadcasts the instruction to relock the plug after a set threshold is expired or if the electrically locking external command is executed by the Gateway.
In the second case the VCU sends the request to close the main switches to the Battery Management System. Once the closure is actuated, the signal to restart the charging process follows the feedback sent by the BMS.

3.1.2 First implemented solution

This section provides a first attempt in representing the aforementioned process, however a quick observation is mandatory on MATLAB configuration parameters. The outcome of the design is a model, which, after a testing and debugging phase, will be used for code generation by means of Embedded Coder. This tool of the Simulink environment requires discrete, fixed step models and a target, i.e. the kind of code, must be selected to generate code for micro-controllers or DSP.

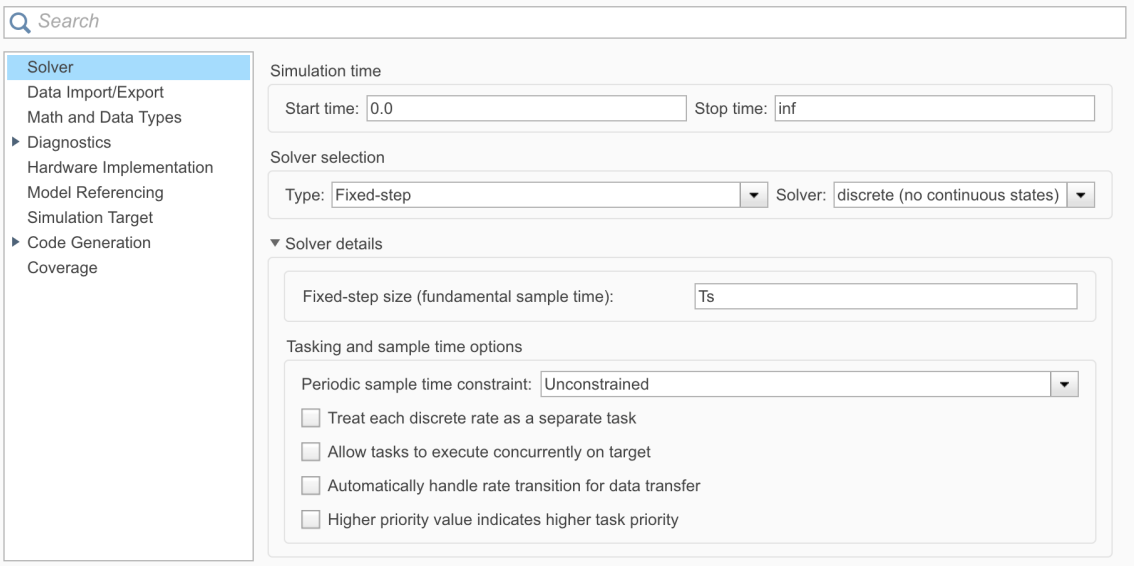


Figure 3.2: Configuration Parameters window from Simulink

In the *Solver selection* of the Figure 3.2, shown above, *Fixed-step* and *discrete* are chosen respectively as type and solver, while the system target file is *ert.tlc* for Embedded Coder, visible after clicking on Code Generation, listed on the left.
The fixed step size is equivalent to the fundamental sample time, which is the lowest if working with multi-rate systems. In this case the fundamental sample time is 1ms, value of Ts, a variable stored in the data dictionary OBC_DD whose DataType is double. The sample time is the same for each block of the Simulink model, in other words the model is single-rate.
Some words need to be dedicated to the signals representation, thoroughly discussed in the previous chapter. The utilization of Bus Objects came from the need of defining the interfaces as structures, due to the boundaries introduced by Stateflow and model referenced blocks, which will be more profoundly analysed afterwards.
For this reason non-virtual buses were used instead of virtual ones.
The big difference between the two is how Simulink treats memory allocation. Virtual buses compile a collection of individual elements, with each passing separately to the system. A non-virtual bus forces the memory to be a contiguous structure passed into the system.
The main reason that comes out for the latter ones is the control of memory allocation in code generation, an important step in the model based design workflow.[1]

The first solution to the problem consists of connecting the two charts, Charger and VCU, and using dashboards to control the input signals to move forward in the simulation. The simulation speed justifies this approach, not compliant with the reference workflow, since it leads to a rapid inspection of system working operation.

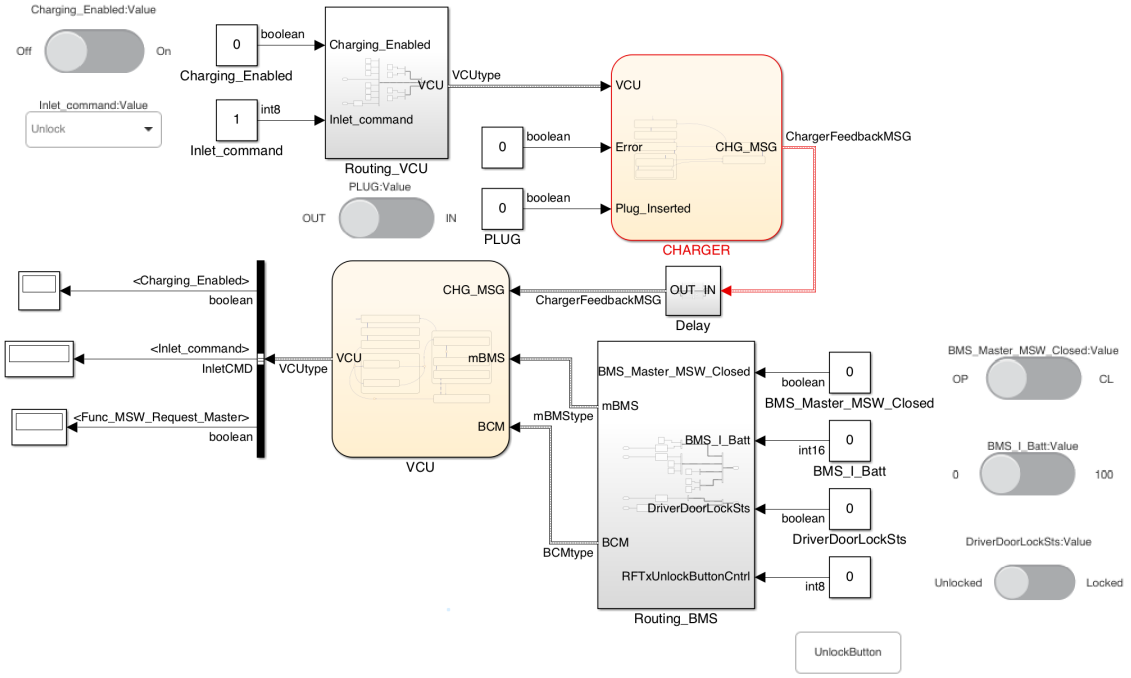


Figure 3.3: Initial top-level model implementation with dashboards

In addition to the Stateflow charts three other subsystems make up the model, i.e. Delay, Routing_VCU and Routing_BMS. The Delay subsystem is placed between the two state machines to introduce a delay in the communication, thus having a model more similar to reality and a remedy to the algebraic loop problem. It consists of a group of unit delay blocks, whose number is equivalent to the number of signals composing the bus CHG_MSG, data type conversion blocks, bus selectors and creators.

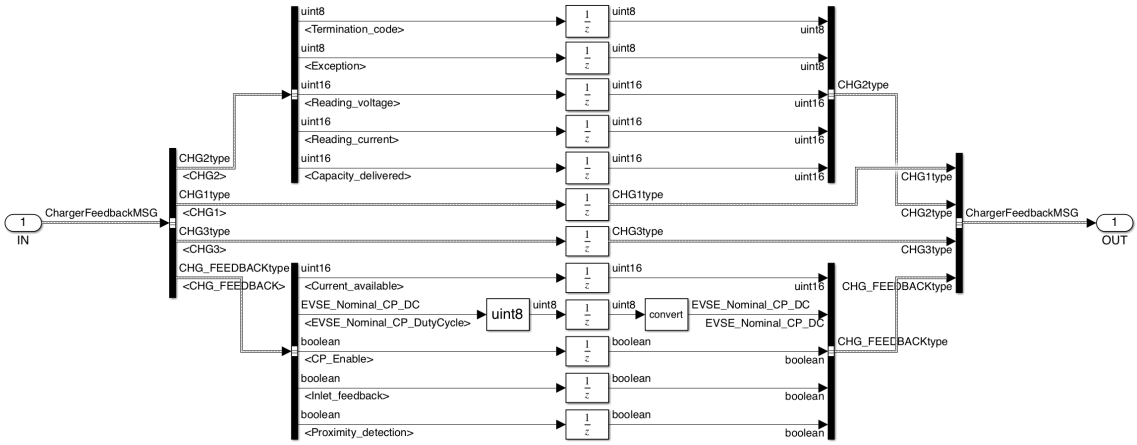


Figure 3.4: Delay subsystem

This block is employed due to the limitation introduced by the bus structure, since some of its signals have different ranges, specifically minimum, and incompatible DataType with the unit delay block, i.e. enumerated.
The other two subsystems allow to control the value of the displayed signals while simulating through the slider switches, the combo box and the push button.

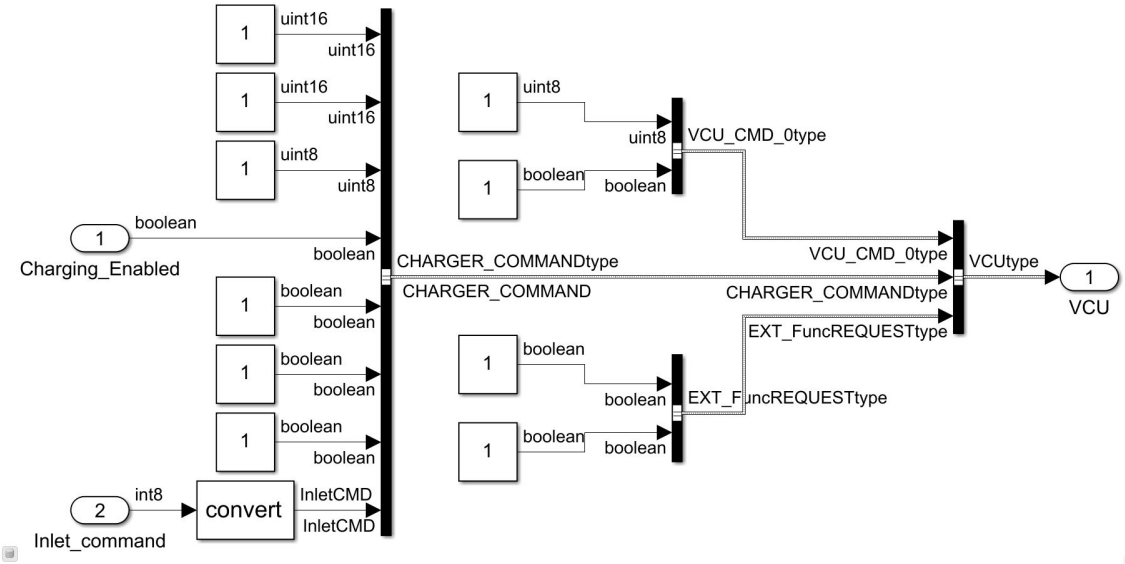


Figure 3.5: Routing_VCU subsystem

It is interesting to explore the Routing_VCU subsystem, Figure 3.5, to have an overview of the VCU bus structure, as it is for the CHG_MSG bus opening the Delay subsystem. The majority of the signals of the VCU bus are out of interest for the inlet lock management and manual stop procedure, which is why they are set to a constant value.
The two input ports are Charging_Enabled and Inlet_command, both signals of one of the three buses composing VCU named CHARGER_COMMAND, whose DataType is respectively boolean and InletCMD (enumerated). As shown in the figure, each arrow connects two blocks. An arrow is a graphical representation of the propagation of a signal whose DataType is displayed above and below it. This function is useful to check and quickly find a DataType mismatch and it is available with other useful features in the debug section of Simulink ribbon menu, listed in information overlays, improving the model readability and debug.

Charger chart

Similar to the VCU, a small part of the charger signals is responsible for the manual stop operation and the inlet lock management.
Keeping in mind that the aforementioned procedure is at the moment the only implemented task, the state machine of the Charger is displayed below in Figure 3.6.

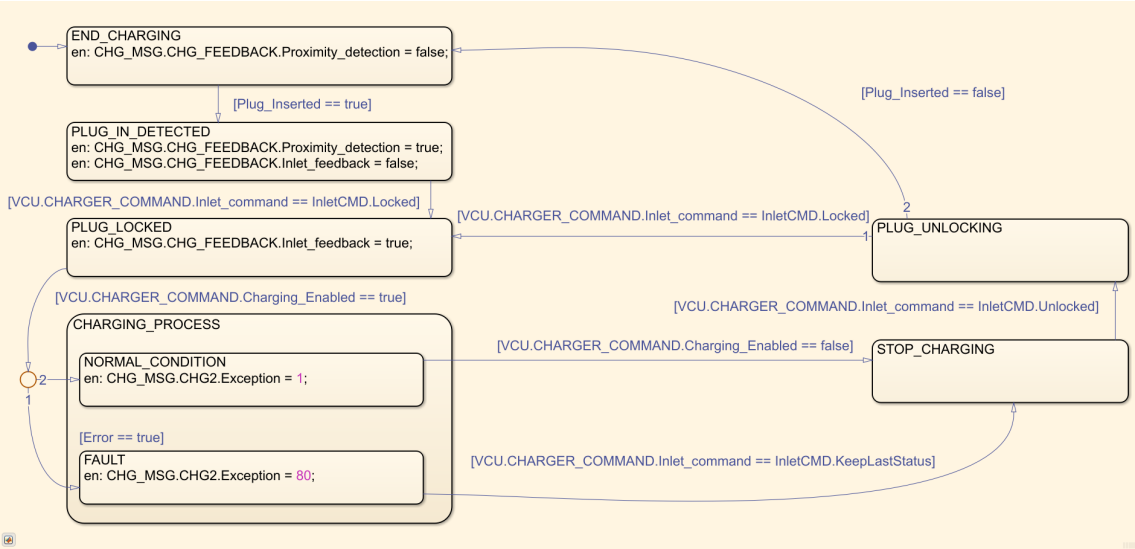


Figure 3.6: First implementation of Charger state machine

It is noteworthy to focus on the signal hierarchy, the choice of using buses led to a structured graphical representation which, despite the names length, guarantees readability. Moreover it speeds up signals management when working with a considerable amount of them, since it is immediate to detect which bus they belong to. The equivalence shown next to a transition, i.e. beside the arrow, expresses the condition to be met to make the state machine evolve. Referring to Figure 3.5 the VCU bus structure appears clear as also its signals position and properties.
The start and the end of charging coincide, indeed END_CHARGING, the default state, corresponds to the charger idling. Charger wakes up after the plug insertion, its presence is communicated on the E-CAN, and it returns to idling after the plug is removed. Its removal can be actuated only after the inlet unlocking.
Subsequent to transition from idle state to PLUG_IN_DETECTED, the state machine entrance in this state forces Proximity_detection and Inlet_feedback respectively to true and false. The first one communicates the plug presence into the inlet, while the latter the inlet status, in this case unlocked thus charging process can not be started. Signals belonging to buses have an extensive name length, to enhance text clarity only the final part is shown to refer to them.
Charger evolves to the next state once hearing from the Vehicle Control Unit the command to lock the inlet. This event corresponds to Inlet_command being equivalent to *Locked*, an enumerated InletCMD variable whose value is the integer two. It is reminded InletCMD class definition is displayed in Figure 2.4.
This is another perfect example of how an higher complexity in structures definition allows to speed up even more the chart understanding. Employing the integer value in the transition condition would have sorted the same effect, but it would have not

shown the significance carried by the number which is quite explanatory through the correct term selection.
The command reception pushes the state machine to move into PLUG_LOCKED, thereupon giving a feedback on the inlet locking. The conditions to start the charging process have been respected and it will begin once, while listening to the E-CAN, Charging_Enable assumes the true value. Together with Inlet_command, they are the only signals employed by the VCU in this procedure. The confirmation to start the charging process leads to a check on the errors presence. The transition is connected to a junction where the value of the input Error is evaluated. This input is a way of representing the detection of a fault by the Charger, e.g. the battery absence. Opting for this design is a modelling choice due to a lack of information in the requirements document. It is only mentioned that Charger communicates its warning or alarm state through the Exception signal transmission, respectively assuming 40 and 80. For simplicity the latter value is used to flag up an error. The battery presence is stated setting Exception to 1.
The state machine evolves in the superstate CHARGING_PROCESS, specifically in FAULT whether an error is detected or in NORMAL_CONDITION. The Charger transits towards STOP_CHARGING independently on which substate is into. However if in FAULT transition happens when the Vehicle Control Unit broadcasts the command to keep the plug locked, otherwise it changes to false the value of Charging_Enabled if the user wants to interrupt the current supply while in NORMAL_CONDITION. This chart does not actually handle the charging process management in line with the assumptions made at the beginning of the paragraph.
When in STOP_CHARGING the Charger next step is to unlock the plug thus it waits for the VCU issuing the order through changing Inlet_command to Unlocked. This leads the state machine to evolve into PLUG_UNLOCKING where it is possible to restart the charging operation, by locking again the plug after the instruction reception sent by the Control Unit, or terminate the process.
The first option occurs once the VCU changes Inlet_command value to Locked, mirroring the user will to restart the charging process, while the latter takes place when the plug is removed.

Vehicle Control Unit chart

A parallel analysis of the Vehicle Control Unit finite state machine helps to have a clearer view of the whole process. It was necessary to project and link the user will to the VCU which has to translate and communicate to the other nodes talking to the CAN. It is important to point out that what is requested is the modelling of the Charger. The VCU was designed for the specific reason of having a deeper understanding, thus dispelling several doubts on Charger design. It then becomes clear that its examination will not be as meticulous as the Charger one.
The state machine displayed in Figure 3.7 progresses by means of the feedback exchanged with not only the Charger but, in addition to it, the Battery Management System and the Gateway, which mirrors the Body Control Module. The BCM signals are useful to the manual stop process only, while the information provided by the BMS is essential for different tasks. Last ones will be examined at a later stage. From the default and idle state, END_CHARGING, to PLUG_INSERTED the machine evolves thanks to a feedback exchange with the Charger. Even if the Charger communicates that the inlet status is locked, the BMS main switches clo-

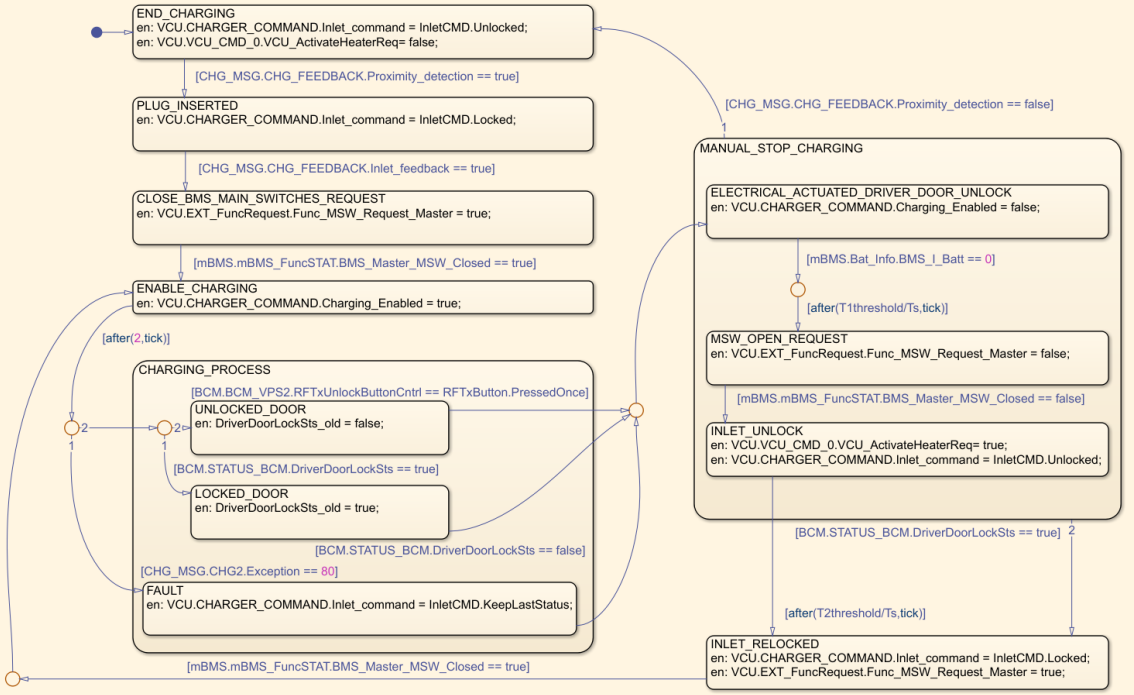


Figure 3.7: First implementation of the VCU state machine

sure precedes the order to start the charging process. A request to accomplish the aforementioned task is sent to the BMS. Its positive reply leads the VCU to enter into ENABLE_CHARGING where it changes the Charging_Enabled value to true. It awaits a set delay to ensure that the Charger checks for possible errors, thus setting the proper Exception value. In normal condition, i.e. Exception equal to 1, it moves into one of the two substates depending on the value of DriverDoorLockSts representing the door lock status. Afterwards the chart evolves as it is thoroughly explained at the beginning of section 3.1.1. Alternately, in fault condition, the VCU commands the Charger to keep the plug locked.
Whether in presence of an error or not, the machine transits into the superstate MANUAL_STOP_CHARGING, where it firstly sends the order to stop the charging process. Initially it waits for the current to zero, information provided by the Battery Management System, thereafter a set time interval. The request to open the main switches to the BMS, as before for the closure, precedes the inlet unlocking, whose order is sent over the E-CAN right after the main switches opening confirmation by the BMS. Section 3.1.1 conclusion goes through the steps that follow the VCU being in INLET_UNLOCK exhaustively.
It is rather curious that both transitions to INLET_RELOCKED due to doors locking and to default state, END_CHARGING, are designed to occur, in any substate of MANUAL_STOP_CHARGING, whenever their conditions are satisfied, but only the first one can do so. The VCU transits to idling when the plug is removed, whose presence is detected by the Charger, in other words once Proximity_detection states its absence. However the plug removal can be actuated exclusively after the inlet unlocking, namely not before the VCU being in INLET_UNLOCK.
This issue is further discussed while analysing the MIL coverage of the final system.

3.1.3 Model testing

This section presents the creation of a test harness for the first implementation of the Charger, examined in an exhaustive way in the previous stage. Furthermore a short coverage analysis will follow to give a preview of the MIL testing.

Model reference

A model can be included in another, called parent model, through a Model block. Each instance of a Model block is a model reference. Employing referenced models comes with several advantages, most of them related to its compiling. It allows to update the model independently from the parent models where it is instanced; to obscure its content to protect intellectual property and, depending on the granted protected-model permissions, it permits to view, simulate, and generate code.
It improves the model loading, simulation speed and code generation. Code is generated only if the referenced model has changed after last code generation. Simulink converts the model to code and runs it to speed up simulation.[11]
To employ an existing model as a referenced one a conversion procedure was followed. It consists of including the first one into a subsystem, which needs to be treated as an atomic system and finally launching the conversion. A set of operations exists and is suggested to prepare the subsystem in order to reduce the possible issues while converting, however they will not be discussed. Nonetheless a check phase is requested by the Model Reference Conversion Advisor, where all the necessary adjustments are listed and could be automatically executed without any control.

Test harness

A testing phase follows the component design. The first step is the generation of a set of stimuli used to derive a collection of output from the device under test.
Simulink handles the construction of a test harness, linked to the model and saved within the project file.

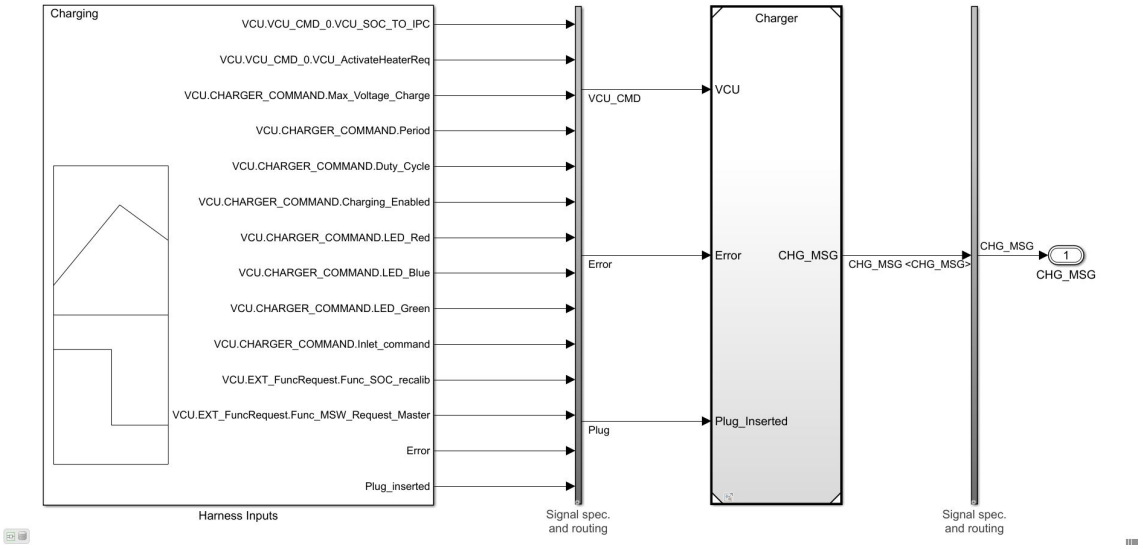


Figure 3.8: Charger test harness

A test harness consists of one or more source blocks that drive the component under test, which, in turn, drives one or more sink blocks.[8]
The two blocks, one preceding and the other one following the DUT, are signal conversion subsystems. These subsystems adapt the signal interface of the source and sink blocks to the graphical interface of the component. The graphical interface of the component includes input signals, output signals, and action, trigger, or enable inputs. The test harness compiles the main model to determine signal attributes, i.e. data type, dimensions and complexity.[8]
The group of input signals is modelled through a signal builder, while the sink is a simple outport which is enough for the task purpose. The component under test is a model reference to Charger.slx, the simulink project file where the finite state machine is connected only to inports and outports.

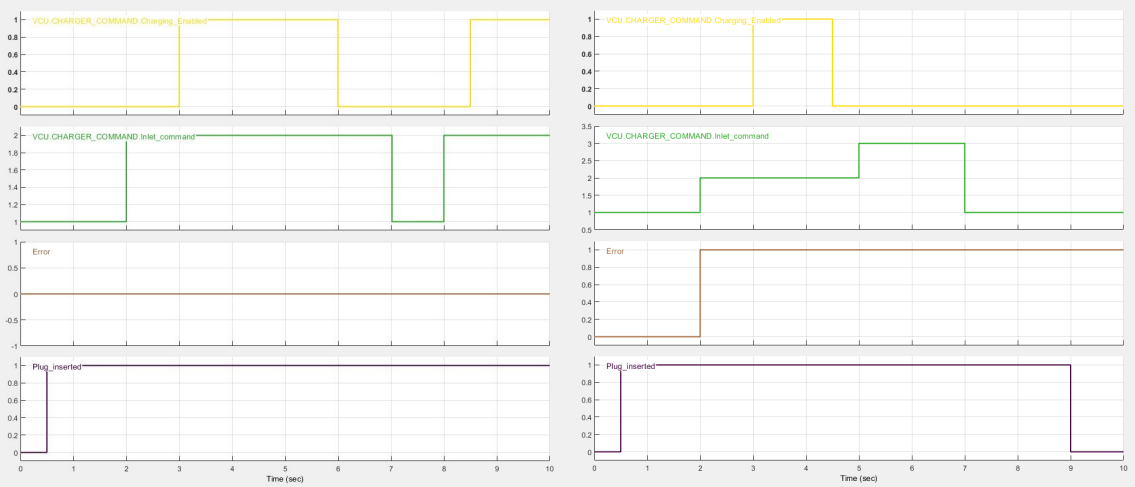


Figure 3.9: Signal builder groups. In order Charging and Fault tests

Figure 3.9 shows two signal groups, each one composing a test to stimulate the component and recreate the working conditions. Both tests last for 10 seconds. Starting from the left, the first group is named after simulating a charging process. Recalling Figure 3.6, the test begins by the plug insertion at 0.5s and it simulates the communication through the E-CAN with the Vehicle Control Unit. At 2s the VCU issues the order to lock the plug, while at 3s the one to start charging.
After three seconds the process is interrupted and at 7s the state machine evolves by means of the command to unlock the plug. The plug is re-locked one second later and the charging process starts again at 8.5s.
Tests are meant to be as close as possible to the real, however a tolerance margin was set on the scheduling timing. The achievement of a complete chart coverage has been imposed as main goal.
The second group recreates a fault condition. The input signals sequence, except for Error whose value switches to true at 2s, is equivalent to the first test until 4.5s where the VCU changes Charging_Enable to false, according to its chart in Figure 3.7. Nevertheless the state machine moves to the next state at 5s through the command to keep the plug locked and after two seconds by means of the unlocking one. The plug removal at 9s takes the Charger back to idling.

Coverage analysis

The importance of coverage analysis is stated by the possible issues coming from code generation. It is necessary at this stage to remind that the outcome of the Model Based workflow is the integration into the target hardware of the code, whose generation is linked to the designed system. The testing phase is conceived to take into account for both intended and unintended behavior before generating code.
Coverage Analyzer was employed to launch the analysis and therefore produce an exhaustive report. Here every transition is examined in an extensive way, providing coverage details such as the percentage related to the adopted metric.
Three different metrics have been used to evaluate the tests effectiveness, Modified Condition/Decision Coverage, Decision Coverage and Condition Coverage.
Following a coverage criteria subsumption hierarchy, the *multiple condition* is the strongest criterion. Stating that a decision is a boolean expression comprising conditions and zero or more boolean operators, the aforementioned criteria requires test cases that cover all the conditions combinations in a decision. Therefore to satisfy it a decision containing n conditions needs at least 2^n test cases, which increases the cost of this criterion, resulting unpractical for large and complex systems. In addition to it some conditions combinations may be unfeasible and filtering them out further increases the cost.[2]
MC/DC on the other hand is a more practical criterion and hence usually a testing requirement for critical systems such as those developed in the avionics domain.
It is satisfied when, in a program, every condition in a decision has taken all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. Therefore the MC/DC criterion is satisfied for a decision, containing n conditions, by a minimal set of $n + 1$ test cases.[2]
Condition and Decision Coverage have no subsumption relationship. To satisfy the first one it is required that each condition takes on both values, while for the latter it is mandatory that every decision has assumed all possible outcomes. It is noteworthy that if a decision contains a single condition they are equivalent.

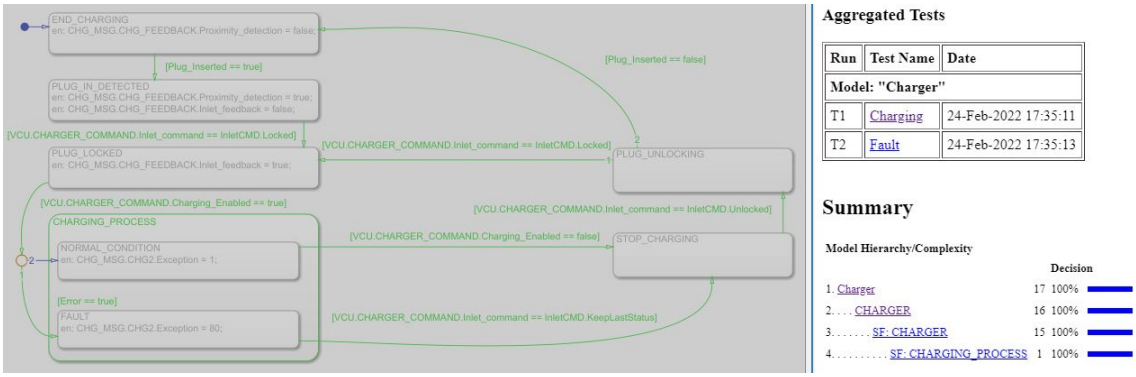


Figure 3.10: Coverage analysis results of the Charger

Since the program in question is a Stateflow chart a transition from a state to another indicates a decision, whose single or more conditions need to be fulfilled in order to be executed. It then becomes clear that the chart in Figure 3.6 can be

evaluated through the Decision Coverage criterion only, which in this case coincides with the Condition Coverage one. Every transition in the state machine is indeed characterised by a single condition.
Coverage Analyzer results show a full coverage, meaning the MIL testing has been performed successfully. The generated report is linked to and ensures traceability with the examined chart. Its states and transitions are denoted by the green colour in Figure 3.10, which demonstrates coverage criterion fulfillment. The uncovered parts would appear in red and by clicking on one of them Simulink traces back to the report page dedicated to that part.
The step following the achievement of full coverage while performing the MIL is the code generation and its coverage analysis. In other words the SIL testing requires to be performed and its results to be compared with the MIL ones. A complete code coverage indicates an accurate description of the model through it, thus ensuring the modeled behaviour. This procedure has been applied to the complete model, thoroughly examined in the following section.

3.2 Complete model

Previously the inlet lock management and the manual stop procedure were examined, followed by an exhaustive study of their representations in the model. In addition to it an introduction to the model testing was provided, together with the explanations of model reference, test harness and coverage analysis.
In this section the Charger model will be updated by the integration of the SOC recalibration procedure, which includes the charging process management. A testing phase comes after with an in-depth focus of the extra parts which finalise the model. At this stage the tool Simulink Test was used to perform MIL and SIL testings. The customisation of the code preceded the conclusion of the code generation process.

SOC recalibration procedure

As previously mentioned, the system designed up to now lacks in a charging process management. The first part of the updated model consists of the execution of a recalibration procedure during the charging phase.
The Vehicle Control Unit node sends over a message to the Enhanced Controller Area Network, addressed to the Battery Management System, in order to start the process, during and at the end of the charging phase. In the first case, discussed in this paragraph, the request is renewed with a set periodicity, which if compliant with the requirements is much longer than the selected time interval of five seconds. The purpose of this choice is to reduce the simulation time, therefore increasing the number of simulations and the quickness in retrieving results.
Each time the VCU requires the recalibration procedure execution, it initially issues the order to stop the charging process to the Charger. The latter assigns the null value to the current set-point and responds to the VCU with a feedback message notifying its standby status.
It waits for a fixed delay to update the VCU with a feedback message, thereupon the VCU sends a request over to the E-CAN to start the recalibration process right after the current reaches the zero value. This information is provided by the BMS. The same node commences the procedure under consideration lasting for a hundred milliseconds.
At this point the requirements document lacks in accuracy, as a matter of fact it does not supply information on how the VCU handles the BMS recalibration. Specifically it is not precised how it proceeds to the next command. To achieve the process the BMS needs a certain time window, therefore, considering the latter as a delay, an option for the Control Unit could have been to move forward once it expires. It was rather opted for an handshake mechanism, namely a feedback sent by the Battery Management System successive to the recalibration completion. BMS_recalibrated is the feedback signal in question.
Subsequently the VCU checks the battery minimum cell temperature, data shared by the BMS, and if above a certain threshold, the charging process can restart issuing the related command. The requirements paper lacks again in precision, indeed it is not described which behaviour has the component if the temperature is below the threshold. In order to fill this lacuna, in case the aforementioned condition occurs the VCU machine enters into a state where it can not exit until the cell temperature overcomes the lower limit. No logic on how to handle a heating process was modeled since it is out of interest.

If the cell temperature is above the lower bound, the Charger reads the VCU order over the CAN, it assigns a value provided by the BMS to the current set-point and it communicates its status, i.e. battery on charge. In this way the charging process can restart.

Full charge

A State Of Charge final recalibration is further required when the Control Unit senses the completion of battery charging, by means of the information supplied by the BMS bus. Within the bus two signals are carrying data, respectively the SOC percentage and the maximum cell tension value.
Given an upper bound for both, whenever one of the two is surpassed, the VCU issues the order to stop the current flow. The Charger listens to it, zeroes the current set-point and transmits a feedback message containing its status. It further notifies the VCU through another feedback message after a fixed delay expiration.
The Battery Management System constantly renews the battery supply current on the E_CAN, whose value reaching zero is the input to the Control Unit for sending the recalibration request to the BMS. As previously specified the process goes on for a hundred milliseconds after which, following a design choice, the BMS communicates the recalibration conclusion.
Next step is a check on minimum cell temperature and if lower than a fixed limit, it was opted for the already presented measure in the antecedent paragraph. Alternately another check is run on the same BMS signals that started the last recalibration process. If the SOC percentage or the maximum cell tension is below a certain level the VCU issues to restart charging. Otherwise the latter sends a request to the BMS to open its main switches.
This action, as already stated in the inlet lock management process, always precedes the plug unlocking that comes before its removal, thereupon the machine idling.

3.2.1 Charger chart

In this stage, the ultimate finite state machine is examined. It consists of the model shown in Figure 3.6 enriched by a charging management process alongside the part dealing with the SOC recalibration procedure and the end of charging.

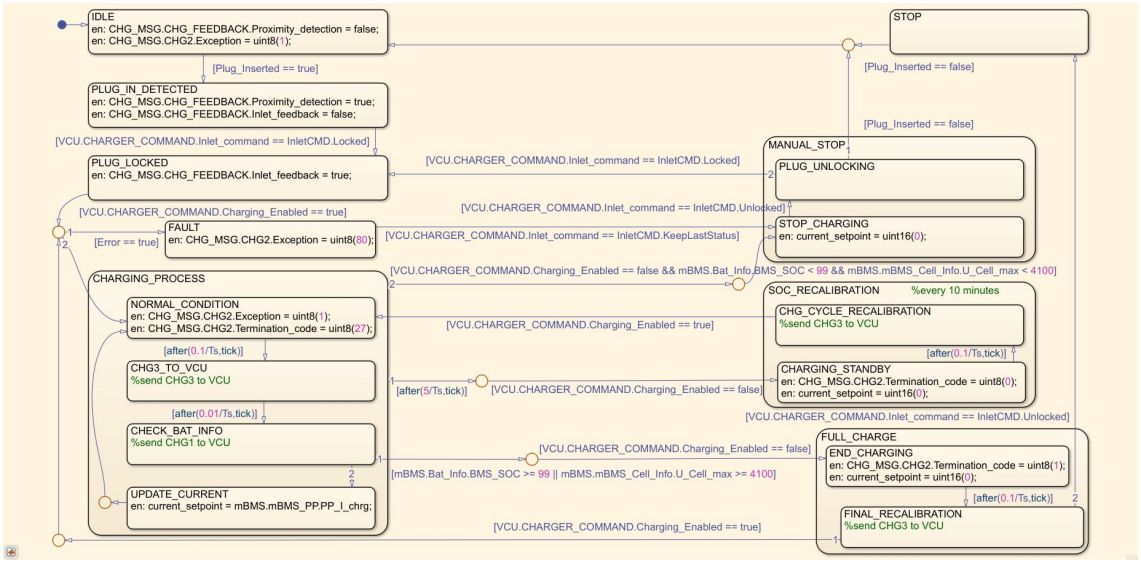


Figure 3.11: Complete Charger finite state machine

Hierarchy in the state-chart development is used to enhance readability by means of a structured layout. In this regard, from a comparison with the initial model, MANUAL_STOP includes two substates which were previously independent. The upper half of Figure 3.11 displays what was analysed in the inlet locking management section, except for FAULT, here excluded from CHARGING_PROCESS, and STOP states. Reminding that the main focus is on the additional part, Charger wakes up from idling after detecting the plug and issues the information to the Vehicle Control Unit changing Proximity_detection value to true, together with Inlet_feedback to false. The latter shares the unlocked plug status.
Once VCU replies by means of the command to lock the plug, i.e. Inlet_command to Locked, the state machine evolves into PLUG_LOCKED where, while instructing the actuator to lock the plug, it updates its status. Subsequent to the Charger feedback reception the VCU orders to start the charging process.
In case of fault, therefore once Charging_Enabled is true and Error is high, the machine evolution from FAULT state on is equivalent to the one presented in the first implemented solution (refer to section 3.1.2).
In case of normal condition Charger transits into a loop in CHARGING_PROCESS, starting from NORMAL_CONDITION where it sets Exception to 1 and Termination_code to 27, meaning respectively battery present and battery on charge.
A hundred milliseconds later Charger evolves to the next state and sends over to the CAN, addressed to the VCU, the feedback message CHG3, namely the same feedback it sends every time the charging process is stopped for periodical and final SOC recalibration. Before transition to CHECK_BAT_INFO the state machine waits for a fixed delay, after which reads over the CAN the information provided by the BMS to check the charging process. Indeed if BMS_SOC or U_Cell_max are above their threshold, Charger stops the process because battery is fully charged

and, once the VCU commands it, it moves into END_CHARGING, substate of FULL_CHARGE. Otherwise the state machine evolves into UPDATE_CURRENT where fixes the current set-point to the value supplied by the BMS, thereupon goes back to NORMAL_CONDITION.
There exist other conditions to make the machine exits from the loop, that are independent on which child state of CHARGING_PROCESS the machine is. The loop is interrupted firstly if the time interval to start the periodic recalibration expires, secondly if the user actuates the manual stop procedure. The two cases are introduced in the same order Charger evaluates them.
It is noteworthy to underline that Charger can interrupts the charging process only if the Control Unit issues the order to do so, therefore changing Charging_Enabled to false. Moreover this command is too general, thus it is not sufficient to differentiate where the machine should move to. To overcome this problem, additional conditions are considered along with the aforementioned one.
Each time the state machine enters into CHARGING_PROCESS a timer begins counting, whose end determines the need of a SOC recalibration. The same happens in parallel in the VCU machine and once it reaches the end it sends the command to make Charger transit to SOC_RECALIBRATION, into CHARGING_STANDBY. In this state the machine updates its standby status by way of Termination_code and zeroes the current set-point to stop the charging. After a hundred milliseconds it sends CHG3 and it awaits the VCU command to restart the charging.
The second condition, only evaluated if the first one is not satisfied, is the manual stop command issued by the user. The Control Unit sends Charging_Enabled equal to false as for the other transitions intended to stop the charging, however it is related to the manual stop whenever the battery is not fully charged. Then if both BMS_SOC and U_Cell_max are below their threshold, once VCU issues the command Charger exits from CHARGING_PROCESS, regardless of any substate it is into, and transits to MANUAL_STOP. Section 3.1.1 deeply illustrates its routine, whose final step is the plug unlocking. The latter is a necessary measure that always precedes the plug removal, indeed it is a possible outcome leading Charger to idling. The other possible transition is the plug relocking to restart the charging process.
The fulfillment of battery charge is the last condition that leads the state machine to interrupt the charging process. Briefly introduced before, Charger reads the message sent by the Battery Management System, i.e. mBMS.Bat_Info whose signals provide information on the battery, each time it transits from CHECK_BAT_INFO. Charger evaluates if either BMS_SOC or U_Cell_max is above its threshold. This condition is contemporaneously evaluated by the Control Unit. As a matter of facts it follows the reception of Charging_Enable change to false by Charger, consequently moving towards FULL_CHARGE, specifically into END_CHARGING.
Subsequently the machine communicates the completion of the charging process by the transmission of Termination_code equal to 1, moreover it zeroes the current set-point compliant with the end of charging. The feedback CHG3 sending follows always the charging suspension or end, therefore a hundred milliseconds after Charger evolves into FINAL_RECALIBRATION. Two possible branches are evaluated. The first one leads to restart the charging process, when Error is low, by the reception of Charging_Enabled to true. If false the machine moves into STOP only after the VCU commands to unlock the plug, Inlet_command equal to Unlocked.
After the plug removal, i.e. Plug_inserted changes to false, Charger returns to IDLE.

3.2.2 VCU Chart

The Vehicle Control Unit has been modeled in order to verify the validity of Charger and to simplify its design. Connecting the two in a feedback loop provides indeed a practical way of simulating, moreover gives evidence of the right timing for tests generation and eases tracking the simulation progress. An higher view of the systems network is shown and shortly examined further in this document.

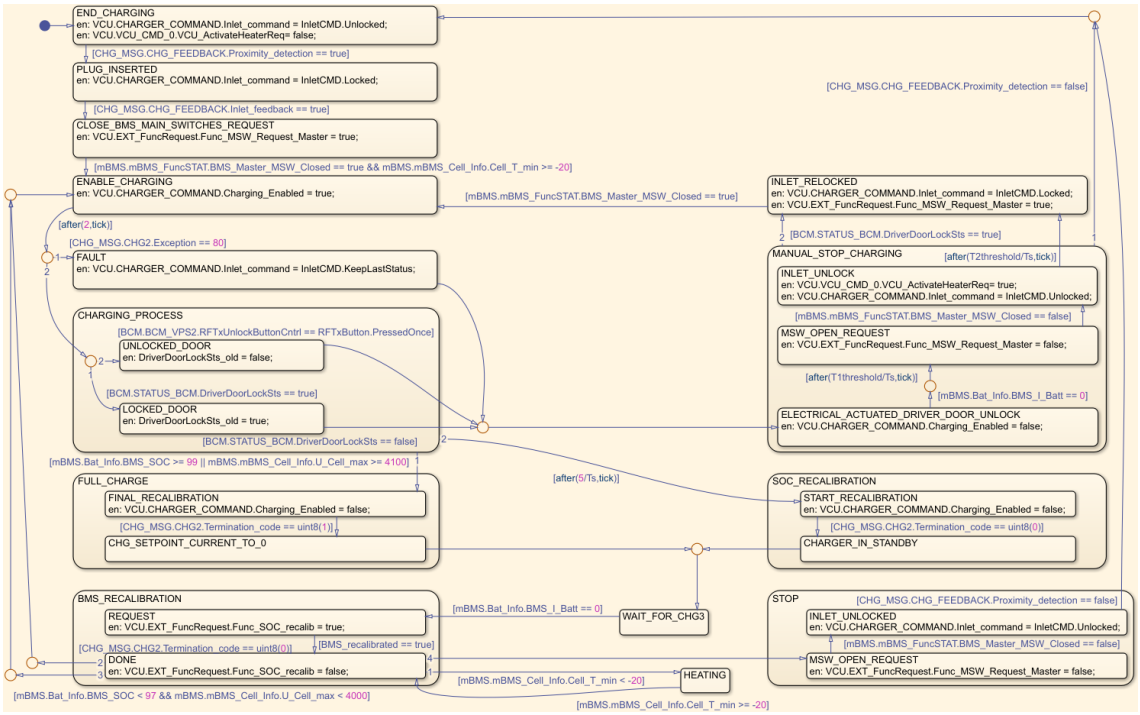


Figure 3.12: Vehicle Control Unit final chart

The chart above presents no changes compared to the one in the upper half of Figure 3.7, except for the extraction of FAULT state from CHARGING_PROCESS as for the Charger state machine. Moreover an additional condition is considered moving from CLOSE_BMS_MAIN_SWITCHES_REQUEST to ENABLE_CHARGING.
The VCU stops idling once Charger informs it about the presence of the plug in the inlet through Proximity_detection signal to true. The Control Unit moves into PLUG_INSERTED where, by means of Inlet_command equal to Locked, instructs for the plug locking the Charger, whose reply is changing Inlet_feedback to true. Its reception makes the VCU evolve into next state where it sends the BMS the request to close its main switches, whose positive feedback was previously enough for next state transition. At this stage it is necessary that the minimum battery cell temperature is above or equal to the lower limit. The fulfillment of this condition leads the state machine to issue Charging_Enable to true to start the charging process.
Next transition happens after a delay of two system ticks. The chart updates with a fixed step equal to the sample time which is set to 1ms. In other words the state machine is delayed of 2ms before evaluating the error presence communicated by the Charger feedback, i.e. Exception. The value 80 indicates an error detection causing a transition to FAULT. A fault event is handled through the already discussed manual stop procedure, similarly to the supertransitions occurrence in both substates of CHARGING_PROCESS, possible only in normal condition.

The communication by the Charger of errors absence pushes the VCU into CHARGING_PROCESS. Regardless of which child state, whose examination is given in section 3.1.2, the machine exits from the parent state if the battery completed the charge or the timer for the periodical SOC recalibration expires.
The first circumstance occurs when either one or both thresholds linked to BMS_SOC and U_Cell_max are overcome. In this case VCU evolves into FULL_CHARGE, specifically in FINAL_RECALIBRATION, commanding the charging interruption. It further moves to the next state once it acknowledges the Charger reception of the order by its feedback Termination_code. The transition that follows is towards a junction which is shared with the alternative branch taken by the machine if the timer expires. This condition is met every ten minutes, shortened to five seconds for simulations enhancements, where VCU transits into START_RECALIBRATION, in SOC_RECALIBRATION, issuing Charging_Enable to false. The same feedback, with different value, is supplied by the Charger to communicate the command reception and execution, therefore the machine is into CHARGER_IN_STANDBY and reaches the aforementioned junction after one sample time interval.
Independently on the followed branch, the VCU moves into WAIT_FOR_CHG3, where CHG3 is the feedback message sent by the Charger each time the charging process is paused or completed. It is opted for not modeling this part since unnecessary for the state chart evolution. The addition of this part, likewise the heating routine mentioned in the SOC recalibration procedure description, is left to who will terminate the customer request.
The BMS recalibration can start only when the charging is interrupted, namely once the current reaches zero. This occurrence, stated by a BMS signal, leads VCU to REQUEST state, into BMS_RECALIBRATION. The machine requires to the Battery Management System to accomplish the recalibration, which on the other side according to a design choice replies through a feedback. Its reception makes the Control Unit transit into DONE where it changes the request signal value to false.
In this state dependently on the covered branch different path can be taken. The first evaluated condition is a check on the minimum battery cell temperature, whose outcome in case of value lower than the threshold was discussed previously. This event may occur regardless of the covered path by the machine.
The second evaluated condition verifies the value of Terminaiton_code sent by the Charger. It assumes zero every time the periodic recalibration must be performed, while one in case of the final one. In other words it is active only for one of the two possible paths covered by the state machine. The third condition is nearly always satisfied whenever the previous one is, however it is designed considering another circumstance. Once the battery is fully charged, hence one of the two BMS signals surpassed its relative threshold, if in the meantime it discharges enough to bring one of those signals below its lower bound, the charging process restarts.
The last transition occurs if none of the other conditions is met. The state chart evolves into MSW_OPEN_REQUEST, whose parent state is STOP. It is requested to the BMS to open its main switches, always preceding the command to unlock the plug, mandatory before its removal. The feedback of the Battery Management System states the request reception and accomplishment, moreover triggers the transition towards INLET_UNLOCKED.
As mentioned before the events sequence is the issuing of the plug unlocking by the VCU to the Charger. The latter provides to instruct the electrical actuator to

perform the unlocking. At the time that the plug is removed the Charger flags it up through changing Proximity_detection to false. It follows the Vehicle Control Unit transition to idling.

3.2.3 Top-level model

A brief analysis is dedicated to the system made up of the two examined charts connected in a loop. Input signals consist of the external ones, Plug and Error, and the ones sent by the remaining nodes in the E-CAN, BMS and Gateway. It is reminded that the latter mirrors the Body Control Module behaviour.

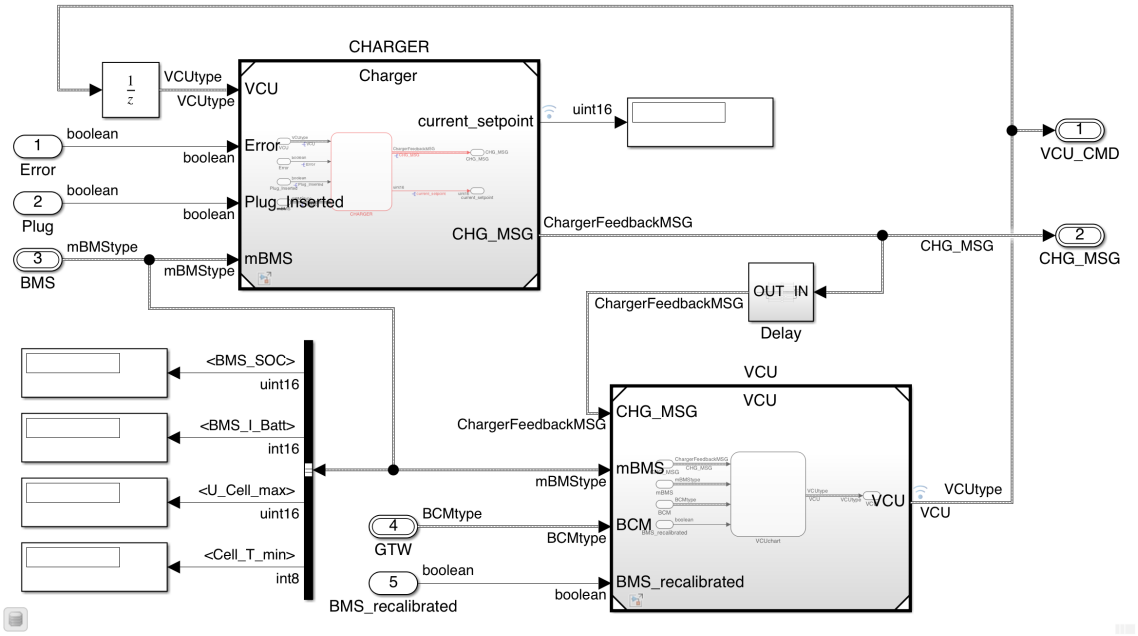


Figure 3.13: Top-level model

The unit delay block and Delay subsystem, whose content is displayed in Figure 3.4, introduce a delay equal to the sample time. It coincides with the fixed step, i.e. 1ms, employed by the solver to perform the simulation. The discrete solver with fixed step selection is mandatory to generate code through Embedded Coder.
While running the model the displays show the evolution of the signals extracted by the BMS bus. Specifically in order, the state of charge indicated in percentage, the supplied current, the maximum tension and minimum temperature among the battery cells.

Testing and coverage

Two signal groups have been used to test the model. An higher number is declared, compliant with the buses definition, however only the ones necessary to stimulate the model are shown in Figure 3.14.
PP_I_chrg is omitted to improve the visibility. Its value, provided by the BMS, is indicative and refers to the maximum current the Charger is able to supply. BMS_I_Batt is modeled taking into account the aforementioned value.
For simplicity a linear trend has been chosen, in addition to the SOC, maximum tension and minimum temperature ones.

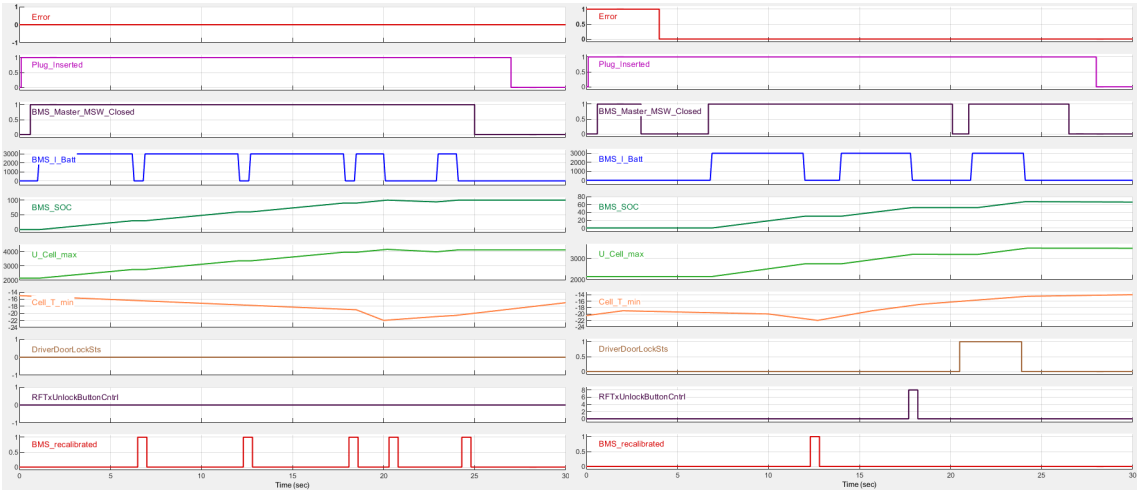


Figure 3.14: In order full charge and fault-relock-manual stop tests

The first test, starting from the left, is a charging process completion. No errors are detected, SOC recalibrations are performed periodically and battery cell minimum temperature goes below the lower bound for a short interval. The test concludes with the full charge achievement, stated by the SOC value surpassing the threshold, and the plug removal.
The second test begins with an error detection, thereupon the plug relock procedure which follows the same steps of the manual stop. The simulation continues through the beginning of the charging process, a heating routine follows a SOC recalibration, and it ends with a manual stop operation.
The combination of the two tests leads to an almost complete coverage of the Charger by means of the Decision Coverage criterion, afterward fulfilled by the addition of coverage filters, further examined at a later stage. A full coverage is reached choosing the remaining metrics for the Charger, while it is not for the VCU. However the accomplishment of this goal is out of the purpose of this activity.

3.2.4 Charger model

The top-level model includes a model reference to the Charger. The latter consists entirely of the state chart shown in Figure 3.11. This section provides the test harness created to stimulate the model and to perform the coverage analysis. In other words the MIL and SIL testings are in-depth examined here and they have been performed by making use of Simulink Test Manager.

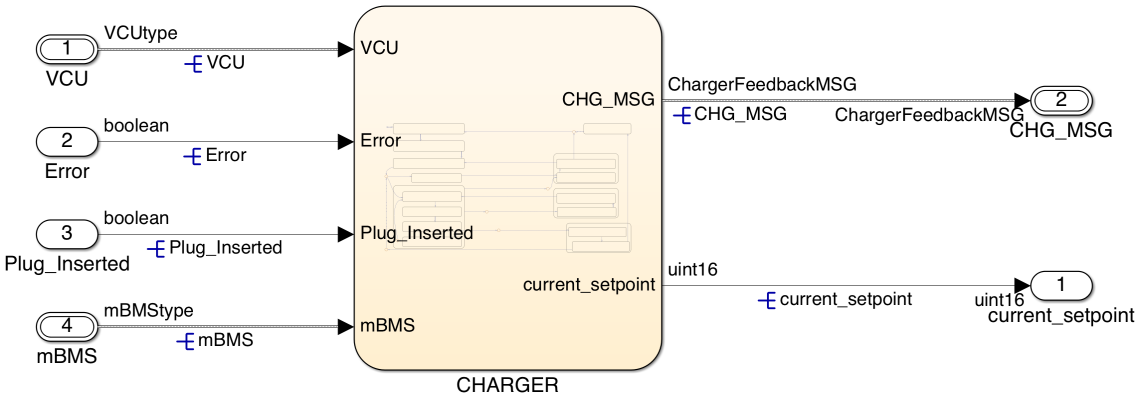


Figure 3.15: Charger model

Clarifications will be provided on details that are useful to accomplish code generation, at a later stage. An introduction to Simulink Check is required since it is mandatory to conform the model to the safety standards ISO 26262, therefore this tool has been employed to acknowledge the proper adjustments to fulfill this task.

MIL testing

The testing phase starts by the definition of a number of signal groups. The created tests are similar to the top-level model ones, indeed the simulation time is the same and they follow the same trend. Specifically the first test simulates a full charging process while the second one consists of a fault detection, followed by a partial charging and ends through a manual stop operation. The missing signals have been adapted to the others previously modeled.

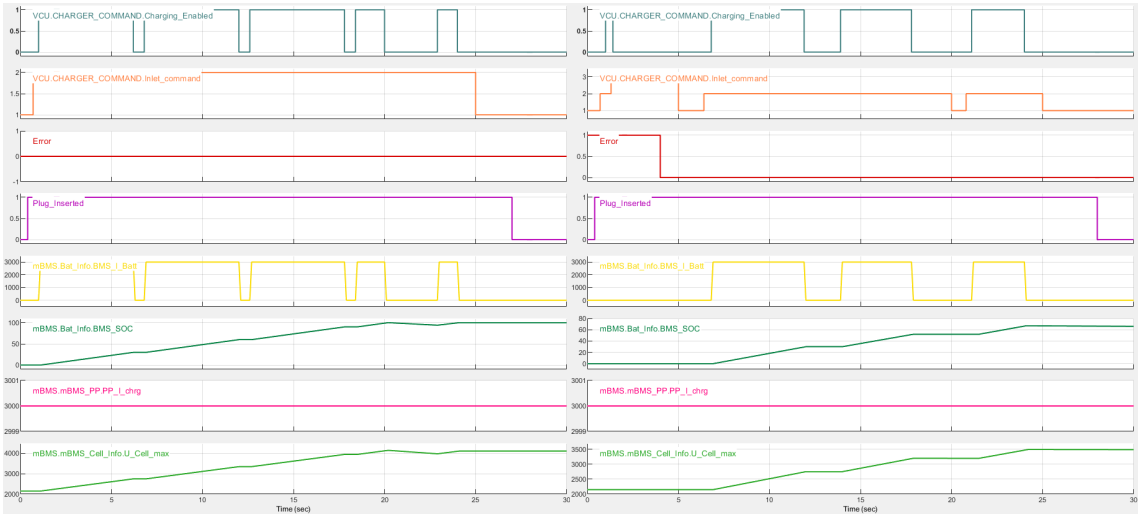


Figure 3.16: Groups of the Charger test harness

This phase has been carried out by means of Simulink Test Manager. Starting from the creation of a test file, a container of test suites, it is possible to choose a test case template. Two simulation test templates have been added.
This test is used to simulate the model, which is inserted in the System Under Test entry along with the test harness. The input of iterations in the dedicated entry follows in case of multiple signal groups. The test harness in question has a signal builder as input, however various inputs are provided while creating a new one.
An item commits to the coverage settings and belongs to the test file, therefore includes all the contained test cases. At this point coverage metrics can be checked, e.g. MCDC, together with recording options and filters can be added.
The filters are applied whenever uncovered links emerge from the analysis. The insertion of a rule lets to justify this event which can be due to a lack of stimuli, which in turn may be linked to the complexity in modeling them.
The last circumstance has been encountered while performing the coverage analysis through the inclusion of the previously presented test harness. In Figure 3.11, the uncovered links refer to CHARGING_PROCESS, specifically to the child states exited when the parent exits. The tests outcomes show NORMAL_CONDITION as the only substate where exits are experienced. A possible explanation is that the machine remains in this state for a longer time when in the loop, due to the set delay. The filter rule addition justifies this lack and leads to a complete coverage of the model. The simulation test allowed to perform the MIL testing.

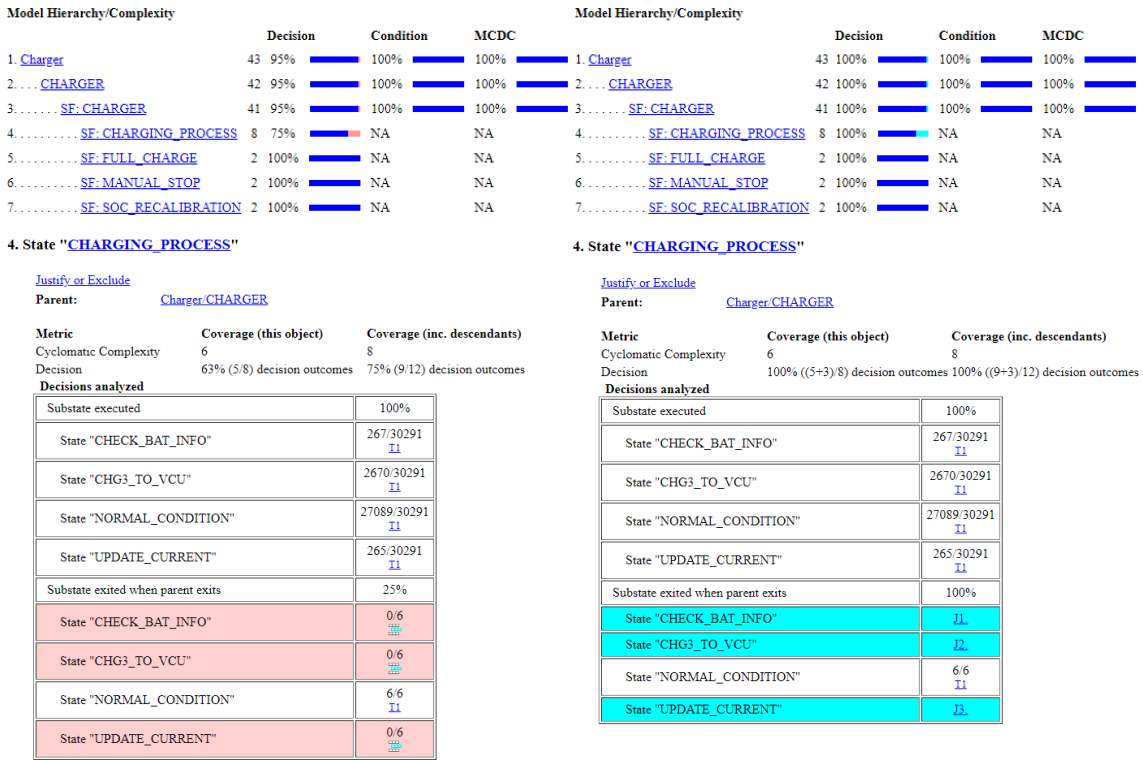


Figure 3.17: Coverage filter application

SIL testing

A full coverage during the MIL testing implies that the model is almost ready for code generation. The last operation is performed once the system is compliant with the ISO 26262 standards. Simulink Check is employed to fulfill this task by running guideline checks and providing the measures to be adopted. At this stage all the suggested adjustments have been applied. Further in this document the aforementioned routine is examined in more detail.
The Software-in-the-Loop testing follows the automatic code generation by Embedded Coder, whose outcome is an object file. Specifically the same inputs, set in the signal builder and employed in the MIL, are used to execute the object code instead of the Simulink model. This phase is preceded by a procedure to prepare and customize the code generation. The next chapter hinges on the aforementioned procedure, while it is now considered a model where these measures have been already taken. Hence the examination focuses on the SIL testing.
By means of the listed signal groups, the test showed the lowest coverage, i.e. 83%, following the MC/DC criterion and did not reach the full coverage following the others. The generated report highlighted the need of an additional test to cope with this issue. The uncovered links are related to the branch connecting FINAL_RECALIBRATION to FAULT and CHECK_BAT_INFO to END_CHARGING. In the previous tests after the charging process completion and the reception of the command to restart it, due to a sufficient decrease of SOC or battery cell tension, an error detection has never been experienced. Moreover every full charge event has been stated by the SOC signal and never by the one monitoring the cell tension.

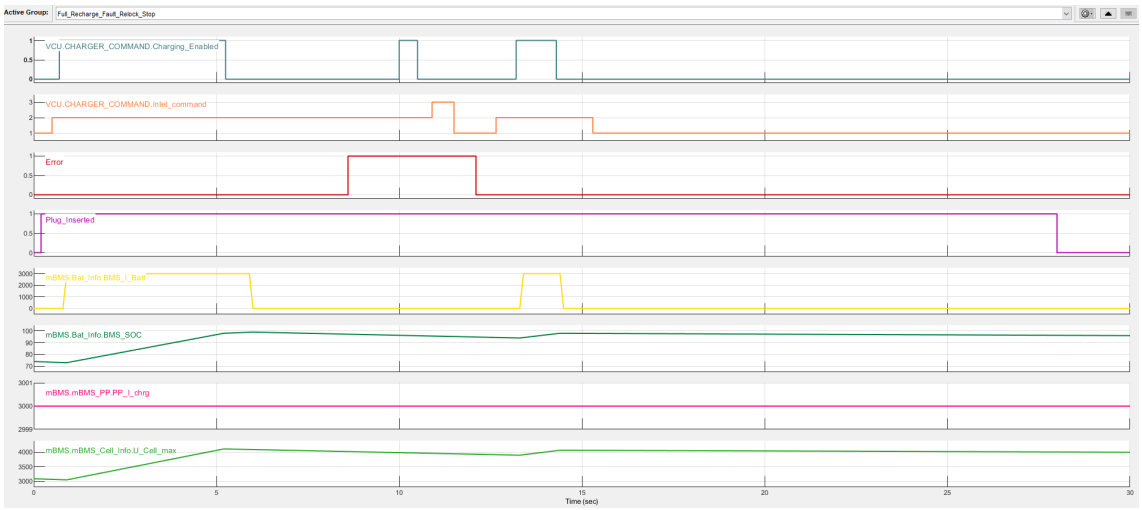
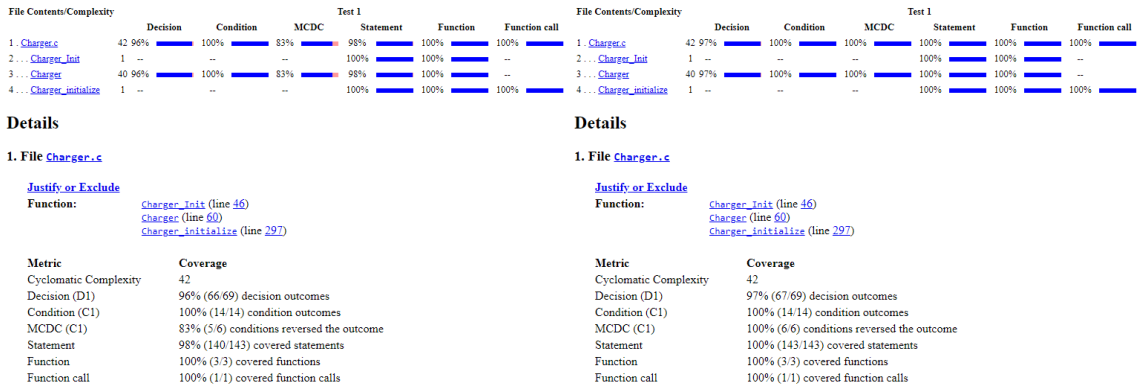


Figure 3.18: Additional test to maximise the code coverage

The test describes a circumstance where the user decides to charge the battery albeit its state of charge is above 70%. Consequently the process ends before the timer to begin the periodic SOC recalibration expires. As a matter of fact, the maximum battery cell tension surpasses its threshold and makes the machine transit to FULL_CHARGE, where it awaits the end of the final BMS' recalibration.
During this time window the charge experiences a decrease and an error is detected. The first one between BMS_SOC and U_Cell_max to go below its threshold triggers the VCU to issue the order to restart the charging process. Due to Error high state,

Charger moves into FAULT and the simulation continues with the stop process leading to a plug relock and, considering that Error returns low, the beginning of a new charging phase. This phase is interrupted by the request to accomplish a manual stop operation after which the plug removal follows and marks the simulation end. The addition of this test to the two already present maximise the code coverage, thus achieving a full coverage when it is opted for the MC/DC criteria.



File Contents/Complexity		Test 1					
		Decision	Condition	MCDC	Statement	Function	Function call
1 . Charger.c	42	96%	100%	83%	98%	100%	100%
2 . . . Charger_Init	1	--	--	--	100%	100%	--
3 . . . Charger	40	96%	100%	83%	98%	100%	--
4 . . . Charger_initialize	1	--	--	--	100%	100%	100%

Details

1. File Charger.c

Justify or Exclude

Function: Charger_Init (line 46)
Charger (line 60)
Charger_initialize (line 297)

Metric	Coverage
Cyclomatic Complexity	42
Decision (D1)	96% (66/69) decision outcomes
Condition (C1)	100% (14/14) condition outcomes
MCDC (C1)	83% (5/6) conditions reversed the outcome
Statement	98% (140/143) covered statements
Function	100% (3/3) covered functions
Function call	100% (1/1) covered function calls

File Contents/Complexity		Test 1					
		Decision	Condition	MCDC	Statement	Function	Function call
1 . Charger.c	42	97%	100%	100%	100%	100%	100%
2 . . . Charger_Init	1	--	--	--	100%	100%	--
3 . . . Charger	40	97%	100%	100%	100%	100%	--
4 . . . Charger_initialize	1	--	--	--	100%	100%	100%

Details

1. File Charger.c

Justify or Exclude

Function: Charger_Init (line 46)
Charger (line 60)
Charger_initialize (line 297)

Metric	Coverage
Cyclomatic Complexity	42
Decision (D1)	97% (67/69) decision outcomes
Condition (C1)	100% (14/14) condition outcomes
MCDC (C1)	100% (6/6) conditions reversed the outcome
Statement	100% (143/143) covered statements
Function	100% (3/3) covered functions
Function call	100% (1/1) covered function calls

Figure 3.19: SIL results before and after test addition

On the other hand Decision coverage is never fulfilled. The right side of Figure 3.19, related to the SIL testing performed after the test addition, shows a percentage close to the maximum.
The use of the temporal logic operator *after* in the Stateflow chart implies the presence in the generated code of indexes, named temporal counters. Their function is keeping track of the events number that triggers the transition guarded by *after*, once the threshold is overtaken.
Two temporal counters are set by Embedded Coder. The first one ending with i_1 takes into account the transitions between two child states, while the second one, i.e. ending with i_2, is related to the transitions whose start point is a parent state. The definition of these two indexes comes along with a condition, one for each of them, that imposes a maximum value that they can assume. In particular the latter is fixed to the maximum unsigned 32 bit integer.
The machine sample time, namely the fixed step after which Charger updates itself, is set to 1ms. At this rate the time needed to overcome the upper limit is more than a thousand hours. It then becomes clear that it is not feasible or rather convenient to set a simulation to cover these two conditions.
The results of the SIL testing can be compared with the MIL ones. This procedure verifies the code behaviour by collecting the outputs, consequences of a set of stimuli. Since the latter is the same employed in the MIL testing the expected results must be equivalent. The equivalence test performs the presented routine, indeed it compares the output of two simulations. The SIL is compared with the MIL, to identify and correct the differences aiming at maximizing the code coverage. The two simulations must be configured in two different entries in the test manager.
Simulation 1 addresses the MIL testing, equivalent to the first set simulation test. Simulation 2 is configured copying the settings from the first one and selecting as simulation mode the Software-in-the-Loop. An additional item is the equivalence criteria, where a tolerance can be selected for each output, i.e. signals of the bus CHG_MSG and current_setpoint.

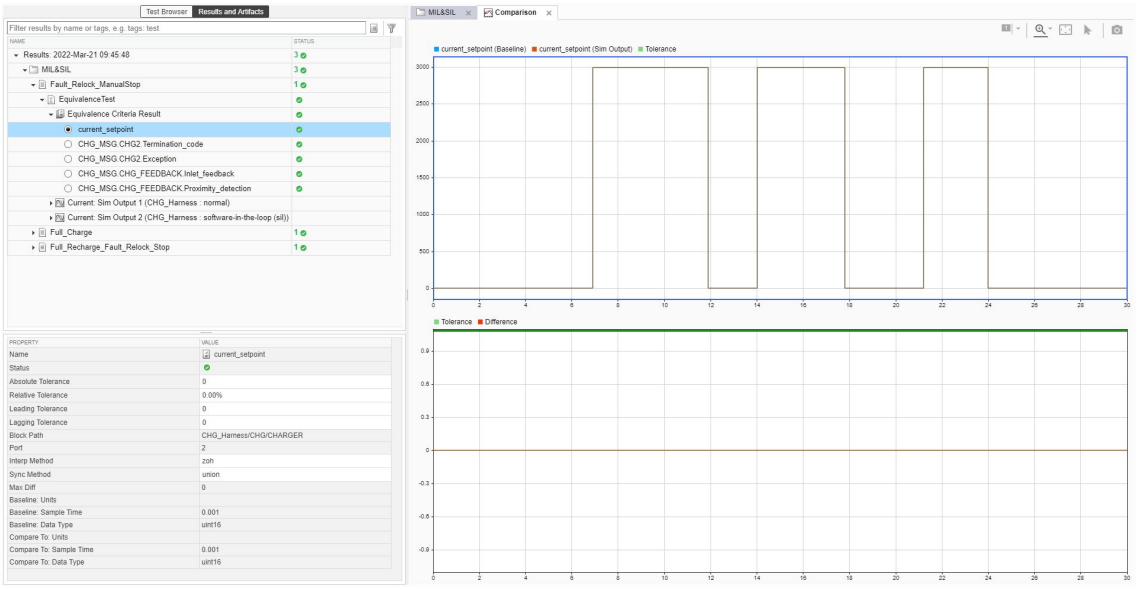


Figure 3.20: Fulfillment of the equivalence criteria for every test

The tolerance is fixed to 0% because the analysed signals are digital.
Figure 3.20 provides a proof of the equivalence criteria accomplishment. It shows for both simulations the trend of current_setpoint on the upper plot and the difference of the two on the lower one, which is zero indeed they are superimposed. The green checks demonstrate the fulfillment of equivalence criteria for every signal of each test, therefore the results of MIL and SIL coincide, implying that the code follows the model behaviour.
Furthermore the SIL testing provides a complete code coverage, stating that the code is valid and can be integrated in the target hardware to execute the tasks it was designed for.

Simulink Check

Simulink Check is employed to prove the safety standards compliance of the model, from which code is generated. Thus adjustments on the model configuration parameters have been applied to conform it, and consequently the code, to the ISO 26262. Specifically this stage may provoke a coverage reduction due to the modifications introduced on the model by the adopted measures. As a matter of fact this operation implied to repeat the testing phase in order to prepare the model for code generation. However the examined model is the final one employed to accomplish this task. A copy of the original configuration set was made, before confirming any change in the parameters, and saved in the data dictionary.

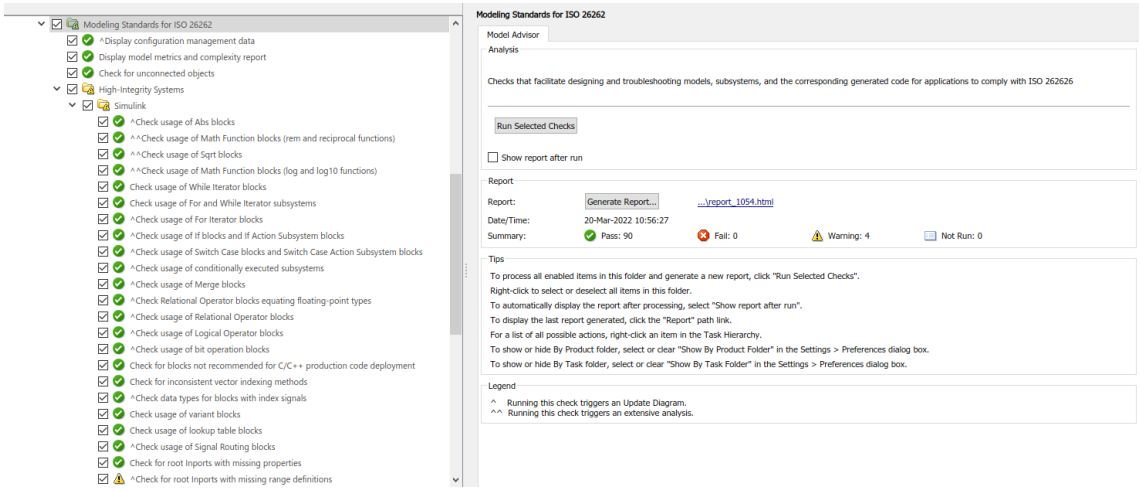


Figure 3.21: Model Advisor window

The guidelines application increases the restrictions and strictness of the model analyser, which flags up any possible cause of non compliance. Above is the result of the examination together with an overview of some checks run over the model.
The picture shows an almost totality of passed checks and few warnings. It is decided to ignore the warnings since some of them are unfeasible and the others are not compromising the model safety. For instance Figure 3.21 lists as last check a warning on the missing range definition for root inports. No range can be defined considering the inports since a part of them consists of buses made up of signals with different ranges. This consideration extends to the outports which represents another warning. By selecting a warning this tool provides the link to the part of the system that needs adjustments, leading to speed up the compliance achievement.
This step is just a part of the verification and validation phase. Its completion allows to obtain a final product whose probability to have faults is minimised. The previously mentioned phase is part of every step of the Model Based Design workflow, e.g the requirements conversion or the testing and coverage analysis. This feature leads to identify bugs and errors and to make modifications while being in the process and not at the conclusion, bringing many advantages like cost decrease and time saving.

Chapter 4

Code generation and integration

The final stage of the workflow consists of integrating the generated code into the target hardware. At the end of the previous chapter, the result of the testings proves the code behaves the same way the designed model does and validates it to proceed with its integration. Taking a step back, code generation and some measures adopted before this process will be illustrated.

4.1 Code generation

Embedded Coder is the tool of the Simulink environment employed to achieve the automatic development of the code, thereupon compiled by the PC compiler into an object file. The SIL was performed by making use of the latter to collect the results by means of the same stimuli used in the MIL.
In the model configuration parameters an entry called *Code Generation* has various subentries, whose names give an idea of their purpose in the process under question.

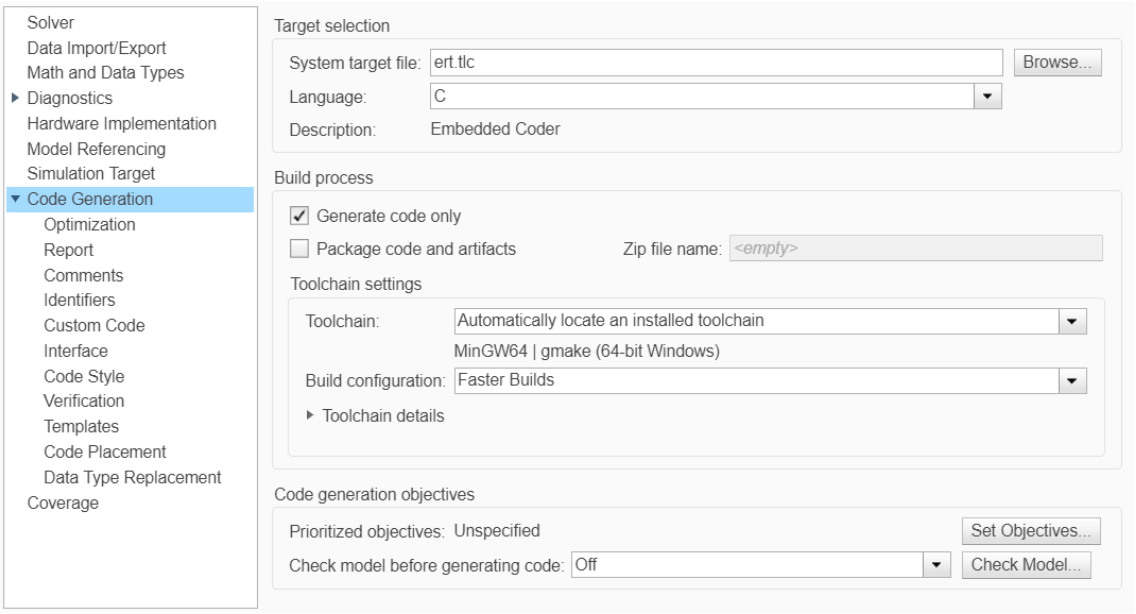


Figure 4.1: Code Generation entry of the model settings

The target selection and build process options are shown above. The first choice is

linked to the solver, in other words to build a model, the model configuration must specify a solver that is compatible with code generation for the system target file. The latter is ert.tlc that supports only fixed-step solver, set in the initial phase of the design process.

TLC files

The file extension refers to Target Language Compiler (TLC). It works with the Simulink software to generate code. A TLC program is a collection of ASCII files called scripts. Because TLC is an interpreted language, there are no object files. The single target file, that calls (with the %include directive) other target files used for the program, is called the entry point.
TLC interprets the set of target files to transform the partial representation of the Simulink model (model.rtw) into target-specific code. Target files provide the user with the flexibility to customize the code generated by the compiler. Using the available system target files allows to produce generic C or C++ code from the Simulink model. This executable code is not platform-specific.
System target files determine the overall framework of code generation. They determine when blocks are executed, how data is logged, and so on. The entire code generation process starts with the single system target file specified in the Configuration Parameters dialog box, on the Code Generation pane.[12]

In the *Build process* section it is possible to disable the code build, add artifacts specifying a zip file by the subsequent check and change below the Toolchain settings. No build happens thus no executable file is generated. Excluding the details, two entries allow to select the compiler, in this case MinGW64, and choose a build configuration.
In the *Code generation objectives* section one or more goals can be set and prioritised while generating code, for instance aiming at improving RAM efficiency. The subitems of Code Generation are entirely dedicated to code customisation.
Referring to Figure 3.15 a symbol similar to a trident stands beside each signal name. It indicates that the signal name must resolve to a Simulink signal object.
Simulink signals have been defined in the data dictionary, namely in its design data. Their characteristics, as DataType or dimensions, must coincide with the signals they are linked to.
Within their definition in the data dictionary at the Code Generation section it is further possible to select the storage class. The presented choices instruct the code on the definition of the variable that mirrors the signal under question. Indeed these variables will be declared with the same name used in the Simulink model, due to resolution. Moreover ExportedGlobal is the storage class to define a variable as global in the code, in order to access to its value from any point of the program.

Generated files

The code generation process produces five relevant files, needed at a later stage dealing with the target hardware. Precisely four header and a source file. An additional source file is generated but solely useful for build process.

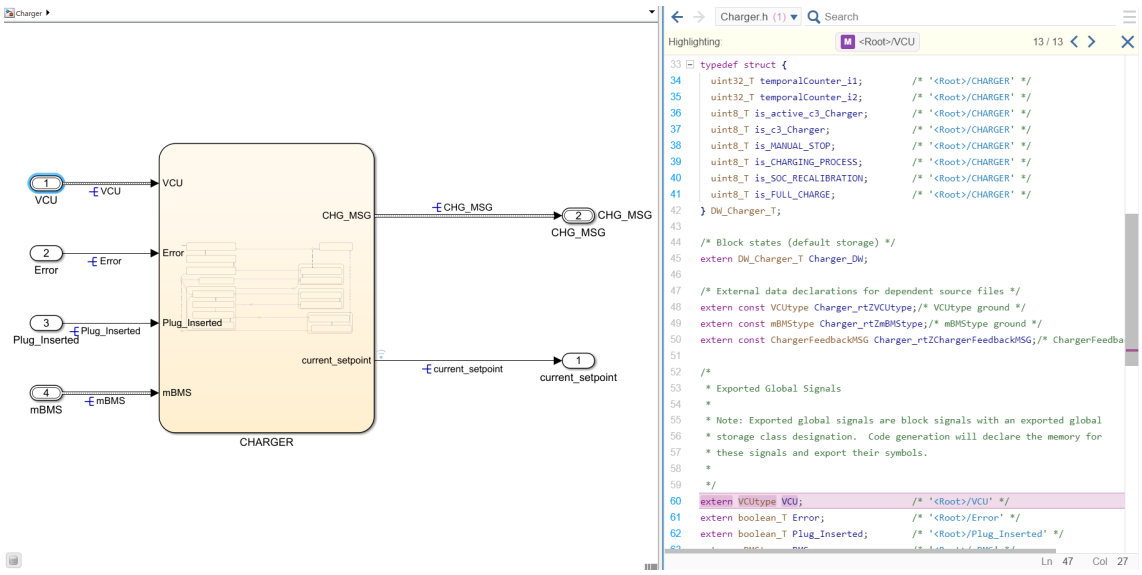


Figure 4.2: Partial header file Charger.h linked to the model

Above is an example of traceability between the code and the model. Figure 4.2 shows Charger.h file partially. The signal VCU is selected in the model and its representation is highlighted in the code. The opposite is possible as well.
The keyword *extern* in the signal definition implies that it is a global variable. Its name coincides with the one of the signal, while its type is VCUtype, related to the bus created and stored in the data dictionary. VCUtype, together with the remaining signal types, is defined in Charger_types.h, included in Charger.h. In the first header file, the buses have been converted into *struct* type definitions. Similarly a type is defined for each Enumerated class and called *enum*.

```
20  #ifndef RTW_HEADER_Charger_h_
21  #define RTW_HEADER_Charger_h_
22  #include <string.h>
23  #ifndef Charger_COMMON_INCLUDES_
24  #define Charger_COMMON_INCLUDES_
25  #include "rtwtypes.h"
26  #endif
28  #include "Charger_types.h"
33  typedef struct { ...
42  } DW_Charger_T;
45  extern DW_Charger_T Charger_DW;
48  extern const VCUtype Charger_rtZVCUtype;
49  extern const mBMStype Charger_rtZmBMStype;
50  extern const ChargerFeedbackMSG Charger_rtZChargerFeedbackMSG;
60  extern VCUtype VCU;
61  extern boolean_T Error;
62  extern boolean_T Plug_Inserted;
63  extern mBMStype mBMS;
64  extern ChargerFeedbackMSG CHG_MSG;
65  extern uint16_T current_setpoint;
68  extern void Charger_initialize(void);
69  extern void Charger_step(void);
92  #endif
```

Figure 4.3: Charger.h without comments

The comments improve the readability of the code, on the other hand they decrease its density. For a more compact code a feature exists to disable the comments, furthermore it is possible to fold the lines of code (LOCs) within the curly brackets.
Specifically Charger.h declares model data structures and a public interface to the model entry-points and data structures.[10] The model entry point functions are Charger_initialize, Charger_step and Charger_terminate. Each one is implemented in the source file Charger.c.
The first one is used for model initialization, generally once. The step function implements the model algorithm, therefore it is usually called every cycle of the while infinite loop. The last one specifies the operations to be done before terminating the program. Its generation has been disabled since it is not needed.
Charger_types.h is included within Charger.h, together with rtwtypes.h. It provides forward declarations for the real-time model data structure and the parameters data structure. It further provides type definitions for user-defined types that the model uses.[10]
The header file rtwtypes.h defines data types, structures, and macros required by the generated code. Often, the generated code requires that integer operations overflow or underflow at specific values. For example, when the code expects a 16-bit integer, the code does not accept an 8-bit or a 32-bit integer type.
The C language does not set a standard for the number of bits in types such as char, int, and others. So, there is no universally accepted data type in C to use for sized-integers. To accommodate this feature of the C language, the generated code uses sized integer types, such as int8_T, uint32_T, and others, which are not standard C types.
In rtwtypes.h, the generated code maps these sized-integer types to the corresponding C keyword base type using information in the Hardware Implementation pane of the configuration parameters.[9]
Every header file has an include statement of this file, as also Charger_private.h. The latter contains local macros and local data that the model and subsystems require [10] therefore it is included in the source file. However in this case it has no content useful for the implemented algorithm.

```
344  /* Model initialize function */
345  void Charger_initialize(void)
346  {
347    /* Registration code */
348
349    /* block I/O */
350
351    /* exported global signals */
352    CHG_MSG = Charger_rtZChargerFeedbackMSG;
353    current_setpoint = 0U;
354
355    /* states (dwork) */
356    (void) memset((void *)&Charger_DW, 0,
357                  sizeof(DW_Charger_T));
358
359    /* external inputs */
360    VCU = Charger_rtZVCUtype;
361    Error = false;
362    Plug_Inserted = false;
363    mBMS = Charger_rtZmBMStype;
364    Charger_DW.is_CHARGING_PROCESS = Charger_IN_NO_ACTIVE_CHILD;
365    Charger_DW.temporalCounter_i2 = 0U;
366    Charger_DW.is_FULL_CHARGE = Charger_IN_NO_ACTIVE_CHILD;
367    Charger_DW.is_MANUAL_STOP = Charger_IN_NO_ACTIVE_CHILD;
368    Charger_DW.is_SOC_RECALIBRATION = Charger_IN_NO_ACTIVE_CHILD;
369    Charger_DW.temporalCounter_i1 = 0U;
370    Charger_DW.is_active_c3_Charger = 0U;
371    Charger_DW.is_c3_Charger = Charger_IN_NO_ACTIVE_CHILD;
372  }
```

Figure 4.4: Charger_initialize function in Charger.c

Figure 4.4 provides the implementation of function Charger_initialize. Both inputs and outputs are set to their initial value, which is zero if not differently specified. At the beginning three constant struct variables are defined and used to initialize the output CHG_MSG bus and the two input buses, VCU and mBMS. The naming convention is imposed on the Identifiers pane, entry of Code Generation in the configuration parameters.
Charger_DW is a struct variable, defined in Charger.h and visible in Figure 4.2, used in the algorithm to keep track of the evolution of the state machine, while the program is running. The finite state machine is converted into code by making use of two switch-case statements and multiple if-else statements.

```
116  void Charger_step(void)
117  {
118    boolean_T guard1 = false;
119    boolean_T tmp;
120    if (Charger_DW.temporalCounter_i1 < MAX_uint32_T) {
121      Charger_DW.temporalCounter_i1++;
122    }
123
124    if (Charger_DW.temporalCounter_i2 < MAX_uint32_T) {
125      Charger_DW.temporalCounter_i2++;
126    }
127
128    if (((uint32_T)Charger_DW.is_active_c3_Charger) == 0U) {
129      Charger_DW.is_active_c3_Charger = 1U;
130      Charger_DW.is_c3_Charger = Charger_IN_IDLE;
131      CHG_MSG.CHG_FEEDBACK.Proximity_detection = false;
132      CHG_MSG.CHG2.Exception = 1U;
133    } else {
134      guard1 = false;
135      switch (Charger_DW.is_c3_Charger) {
136       case Charger_IN_CHARGING_PROCESS:
137        if (Charger_DW.temporalCounter_i2 >= ((uint32_T)((real_T)(5.0 / 0.001))))
138        {
139          if (!VCU.CHARGER_COMMAND.Charging_Enabled) {
140            Charger_DW.is_CHARGING_PROCESS = Charger_IN_NO_ACTIVE_CHILD;
141            Charger_DW.is_c3_Charger = Charger_IN_SOC_RECALIBRATION;
142
143            /* every 10 minutes */
144            Charger_DW.is_SOC_RECALIBRATION = Charger_IN_CHARGING_STANDBY;
145            Charger_DW.temporalCounter_i1 = 0U;
146            CHG_MSG.CHG2.Termination_code = 0U;
147            current_setpoint = 0U;
148          } else {
149            guard1 = true;
150          }
151        } else {
152          guard1 = true;
153        }
154        break;
155
156       case Charger_IN_FAULT:
157        if (VCU.CHARGER_COMMAND.Inlet_command == KeepLastStatus) {
```

Figure 4.5: First LOC of the Charger_step function implementation in Charger.c

When the program reaches the LOC where Charger_step is called, it enters into it and declare two *boolean_T* local variables. This type is defined in rtwtypes.h and it refers to an unsigned char, thus 8 bits are used for boolean_T variables. The first one, guard1, accounts for the program being in the CHARGING_PROCESS loop, in other words for transitions among its child states guard1 is true. This variable is the guard condition to activate the if branch where the aforementioned transitions happen. The latter, tmp, is used to improve code compactness in the branch active when guard1 is true.
Two checks follow from line 120 to 126, previously examined in the code coverage analysis. Upper limits are set on the indexes used to track the time intervals to wait for some transitions. Refer to SIL testing paragraph for detailed information.
The next if statement evaluates a condition that is true only for the first function call. Namely it checks for state machine status, if inactive the program changes the status of the machine to active. Furthermore it assigns to the member (called *position* during code examination) of Charger_DW, responsible for tracking which state, excluding child ones, the machine is into, the value linked to the default state,

i.e. IDLE. Subsequently the program executes the assignments declared in IDLE.
In case the condition within the if statement in line 128 is not verified, in other words the function call rate is higher than one, the program enters into the else branch. After setting guard1 to false, the switch construct is used to move within the state diagram. The program evaluates *position* value to find out which case statement it has to go into, depending on the state the machine is into.
Supposing the program was in FINAL_RECALIBRATION or PLUG_LOCKED case and the conditions guarding the transitions to enter into CHARGING_PROCESS are satisfied, *position* will assume the value to make the program move to line 137. The if statement at this LOC checks if the timer after which starting the periodic SOC recalibration has expired, namely the first prioritized transition when Charger is in CHARGING_PROCESS.
Once the counter i_2 reached the value on the right hand of the expression, Charging_Enabled value is assessed. The expression passed to the if function is true in case the aforementioned value is false, thus the program moves forward to the deactivation of the parent state the machine was into, by changing the member of Charger_DW struct which is tracking its status. For each parent state there is a dedicated member of the struct. An update of *position* follows together with the member value linked to SOC_RECALIBRATION parent state, specifying the child state the machine is into. Subsequently the counter i_1 is zeroed, used for the next transition that happens after a fixed time interval, moreover the assignments declared in this state are executed.
Whenever the program assesses Charging_Enable at line 139 while its value is true, it jumps at line 148 where the else statement is executed and the true value is allocated in guard1. The same happens while evaluating the expression at line 137, if not verified the timer has not expired yet thus the program goes at line 151, enters into the else branch and assigns true to guard1. The following instruction is a break, which leads the program to exit from the switch and to execute the next instruction.

```
285  if (guard1) {
286    tmp = !VCU.CHARGER_COMMAND.Charging_Enabled;
287    if ((tmp && (((int32_T)mBMS.Bat_Info.BMS_SOC) < 99)) && (((int32_T)
288         mBMS.mBMS_Cell_Info.U_Cell_max) < 4100)) {
289      Charger_DW.is_CHARGING_PROCESS = Charger_IN_NO_ACTIVE_CHILD;
290      Charger_DW.is_c3_Charger = Charger_IN_MANUAL_STOP;
291      Charger_DW.is_MANUAL_STOP = Charger_IN_STOP_CHARGING;
292      current_setpoint = 0U;
293    } else {
294      switch (Charger_DW.is_CHARGING_PROCESS) {
295       case Charger_IN_CHECK_BAT_INFO:
296        if (((((int32_T)mBMS.Bat_Info.BMS_SOC) >= 99) || (((int32_T)
297             mBMS.mBMS_Cell_Info.U_Cell_max) >= 4100)) && tmp) {
298          Charger_DW.is_CHARGING_PROCESS = Charger_IN_NO_ACTIVE_CHILD;
299          Charger_DW.is_c3_Charger = Charger_IN_FULL_CHARGE;
300          Charger_DW.is_FULL_CHARGE = Charger_IN_END_CHARGING;
301          Charger_DW.temporalCounter_i1 = 0U;
302          CHG_MSG.CHG2.Termination_code = 1U;
303          current_setpoint = 0U;
304        } else {
305          Charger_DW.is_CHARGING_PROCESS = Charger_IN_UPDATE_CURRENT;
306          current_setpoint = mBMS.mBMS_PP.PP_I_chrg;
307        }
308        break;
309
310       case Charger_IN_CHG3_TO_VCU:
311        if (Charger_DW.temporalCounter_i1 >= ((uint32_T)((real_T)(0.01 / 0.001))))
312        {
313          Charger_DW.is_CHARGING_PROCESS = Charger_IN_CHECK_BAT_INFO;
314
315          /* send CHG1 to VCU */
316        } else {
317          /* send CHG3 to VCU */
318        }
319        break;
320
321       case Charger_IN_NORMAL_CONDITION:
```

Figure 4.6: If statement to deal with the CHARGING_PROCESS loop

The latter corresponds to an if statement whose passed expression is checking if guard1 is true. The condition of having guard1 high is equivalent in the state diagram to evaluate the second prioritized transition, namely a check on the manual stop request. If conditions are not met the consecutive transition depends on which child state of the loop the machine is into. The switch construct is employed again passing the member of Charger_DW struct that tracks the machine position into the charging loop, different from *position* that does not track any child state. CHARGING_PROCESS is the parent state with the highest number of child ones. The others have only two substates, leading the code generator to opt for an if-else construct instead of the switch-case. Considering the main switch function, inside the cases related to the aforementioned parent states, i.e. SOC_RECALIBRATION, MANUAL_STOP and FULL_CHARGE, the if-else statements are used to evaluate which one of the two child states the machine is into. Depending on that, the program executes the assigned instructions.

4.2 Code integration

This last stage is additional indeed it was not initially planned. Nevertheless it has been accomplished because of a favourable time management. This test is purely demonstrative, hence no standard or certified methods have been followed. The goal is to implement an operating routine mostly compliant with the one presented in this document.
The model entry point functions have been exploited to test the generated files on a STM32 Nucleo-144 board. The board provides a STM32 microcontroller (MCU) whose core is an ARM Cortex-M4 at 180MHz, three user LEDs, two user and reset push-buttons, an USB with Micro-AB as board connector and power supply, a 32.768 kHz crystal oscillator, and an on-board ST-LINK debugger/programmer.
Furthermore it provides a comprehensive free software libraries and support of a wide choice of Integrated Development Environments (IDEs), including STM32CubeIDE [6] employed to program the MCU.

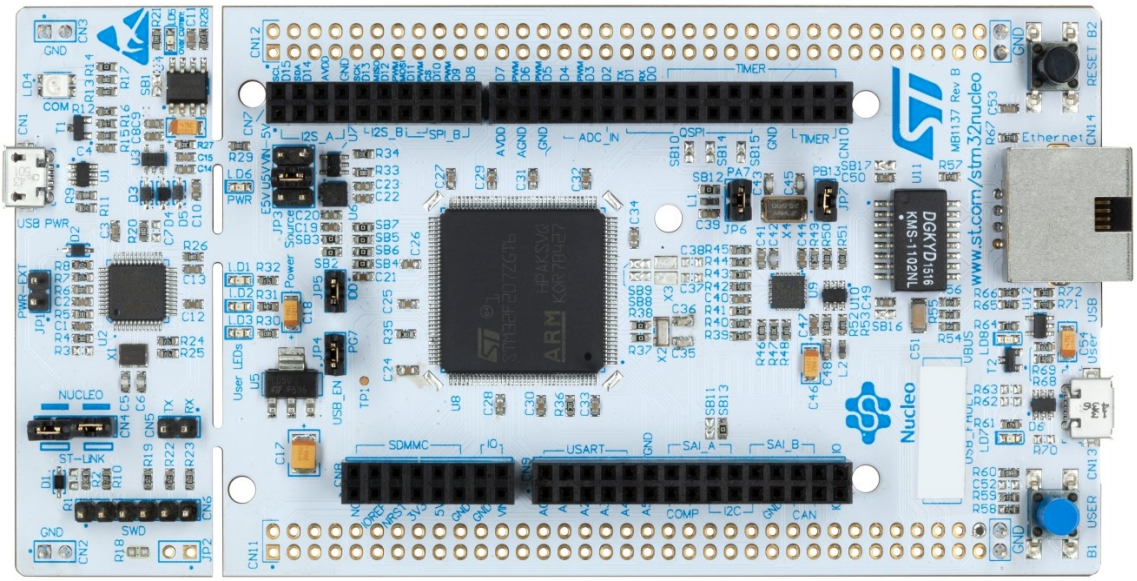


Figure 4.7: STM32F439ZIT6 Nucleo-144 board

The handwritten code aims at calling Charger_step function, defined in Charger.c, at a fixed rate. Therefore since between two different function calls there is a defined pace the state machine evolves in a fixed-step way. It is noteworthy to precise that the highest frequency is limited by the time needed by the MCU to execute every instruction between a Charger_step call and the following one, including the execution time of the function itself. If an higher frequency is set the program assumes an unwanted behaviour.
The idea is to simulate a charging process, thus sending inputs to the model, making use of the available General Purpose Input/Output (GPIO) of the board and coding the remaining ones. The LEDs have been used for a debug purpose, in order to keep track of the machine evolution. It has been earmarked the user button for simulating the plug insertion and removal, to flag up a manual stop request and to address when the path followed by the machine branches out.
Namely in PLUG_UNLOCKING and in FINAL_RECALIBRATION Charger can, generally restart the charging process or move to an idling state.
A single pressure of the user button instructs the code in the first case simulating the plug removal, in the second issuing the command to unlock the plug. Subsequently, since machine is in STOP, another pressure simulates the plug removal.
Multiple pressures within an established time window respectively commands to relock the plug and, if in FINAL_RECALIBRATION, to restart the charging process.
The other inputs, precisely the ones sent by the Vehicle Control Unit and the Battery Management System, are programmed to be automatically set depending on Charger machine evolution. No fault condition is evaluated in the test due to the complexity of finding a strategy to employ the same user button or to code an error detection.

STM32CubeIDE

The first step is the creation of a new project that comes along with the target selection. The device in question is a STM32F439ZIT6 in LQFP144 package. The project setup, consisting of options selection as target language (C), follows with the target's firmware setup.

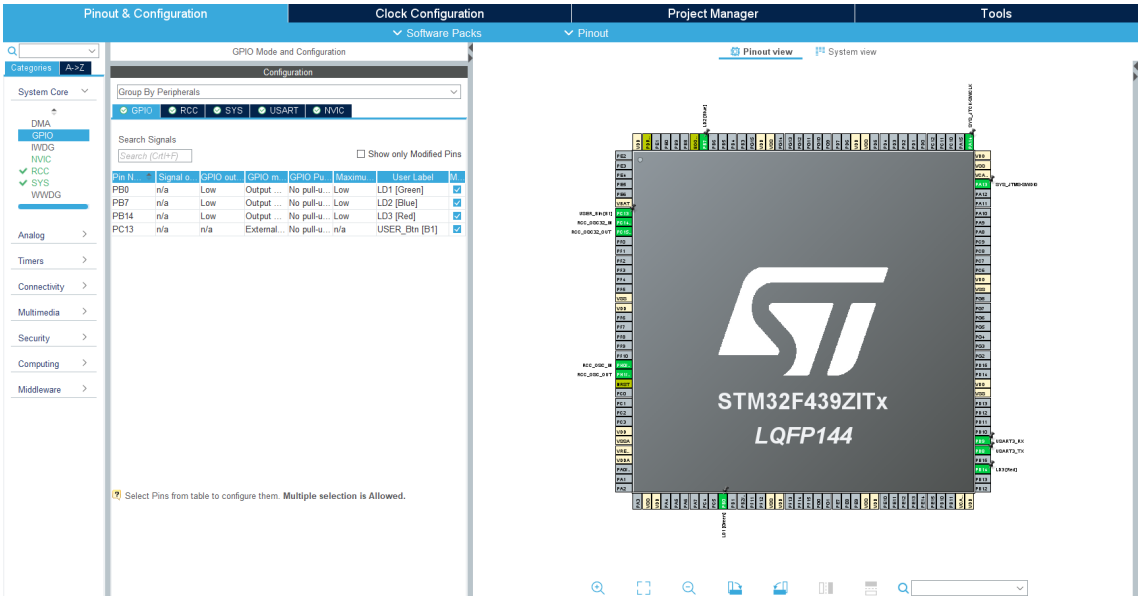


Figure 4.8: Device configuration file window

Next step is the device configuration. A separate file having the same name of the project but different extension, i.e. IOC, provides the target pinout view. From this pane it is possible to choose directly the state of a particular pin or to activate a peripheral, by selecting a working mode. In this case the configuration tool deals with the state of the related pins automatically.
In Figure 4.8 on the left all the components are listed in categories or alphabetic order. For instance GPIO is picked up from the System Core list and the Model and Configuration pane shows the pins connected to the LEDs and the user button. The toolbar allows to visualise the other configured pins relative to different categories, e.g. the reset and clock controller, system, usart or interrupts controller.
Above is the main toolbar to switch to clock configuration, code generation settings within project manager, and more advanced tools. System clock is set to 16MHz.
At the end of the configuration code is generated providing a source file, named main.c, where peripherals initialization, clock configuration and error handling functions are already implemented and most of them placed in the main function.

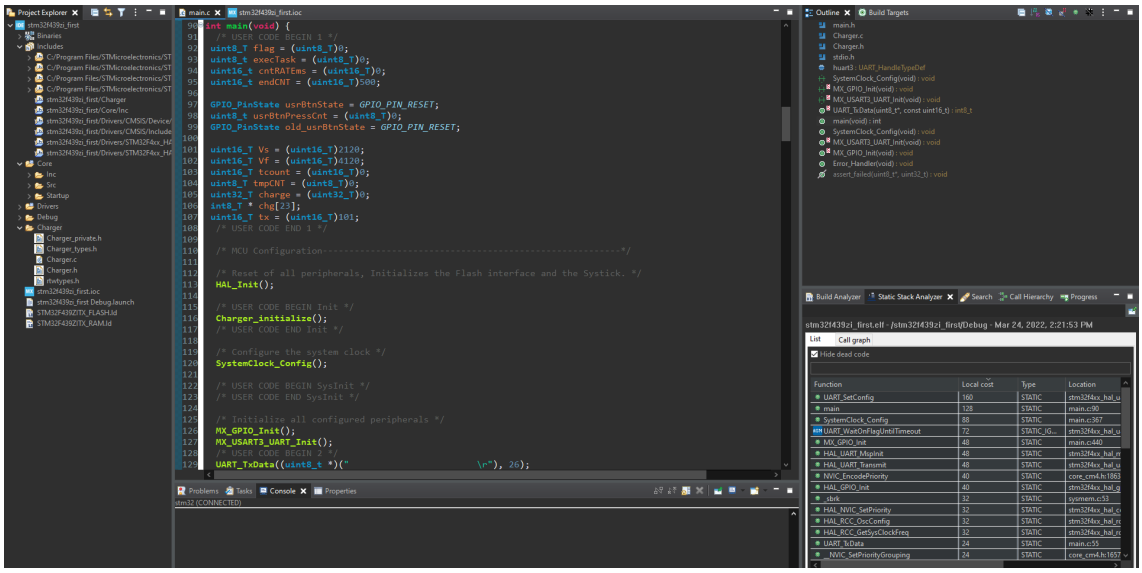


Figure 4.9: STM32CubeIDE code perspective window

The IDE provides an explorer located on the left in Figure 4.9 to access to any file within the project. The Charger folder has been added to the project since some of the generated files are included in main.c. However to access to them it is mandatory to include the directory path within the project. The outcome is the path presence in the Includes section, second entry below the project name in the explorer.
Surrounding the code editor, three windows display information depending on the selected pane in the relative toolbar. The one at the bottom prompts errors and warnings in the Problem pane, while in Console it prints the logs if the global console is selected. In the figure a command shell shows the data transmitted by the MCU to the PC received at the serial port by which the connection is established.
At the top right the listed items are linked to the code and they ease the navigation through the code. At the bottom right a tool to monitor the stack filling.
The supplied template includes comments which make the code more readable and mark areas within the user should add lines. The latter is mandatory in case modifications are applied to the configuration file. Indeed to apply the changes, the

code must be generated again and the IDE is instructed to not overwrite the LOCs included in the aforementioned areas.
The main.c file begins with the include commands of the header files plus Charger.c, where model entry functions are developed. A global variable is subsequently declared, whose type is a struct defined in a built-in library which refers to the hardware abstraction layer (HAL). This library hosts variables and functions to interface the algorithm with peripherals, e.g. to manage the LEDs routine.
The declarations of the initializing functions prototypes follow along with the implementation of a function to transmit over the configured UART. A pointer to a string and the length of the string are passed to this function.
Next lines consist of the main function development. It begins with local variables declaration, followed by the initialisation and clock configuration functions call along with sending a string of blank spaces over to the UART.

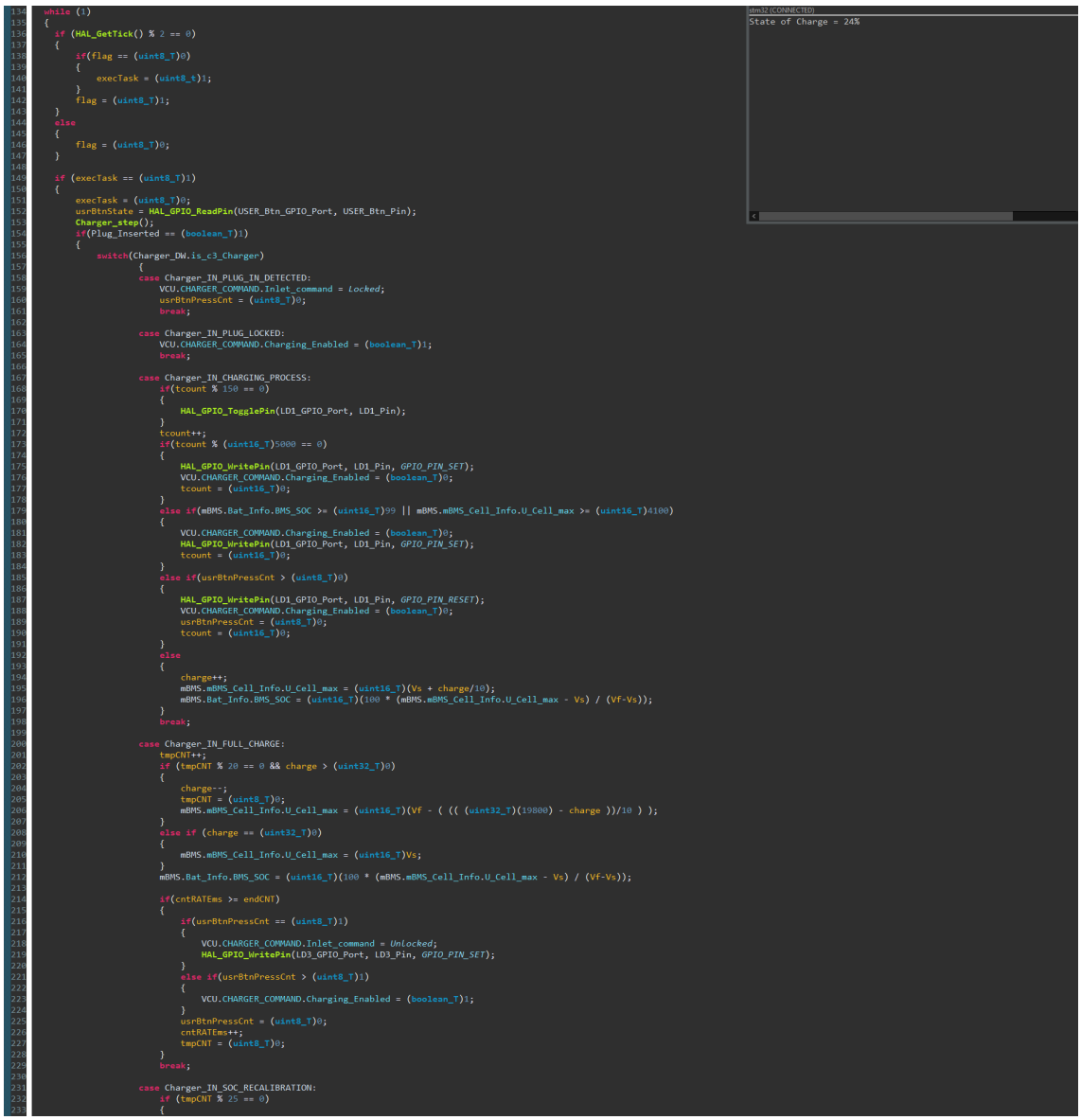


Figure 4.10: Beginning of the while loop

Successively the infinite while loop starts. The implemented routine is active only when condition within the if statement at line 149 is verified. It requires *execTask*

to be true and its value is forced to one each two milliseconds. This check is made in the first if construct by means of HAL_GetTick function which retrieves the system tick. It is not put equal to one millisecond because two is the lowest value below which the program assumes an unwanted behaviour. Furthermore a flag is used to ensure the execution frequency is respected.
Every two milliseconds the program sets *execTask* to zero, reads and stores the state of the pin connected to the user button into *usrBtnState* and forwardly calls Charger_step. No arguments are passed to this function since the variables declared within it are global, thus it is possible to access to them even outside the function. Line 154 checks Plug_Inserted input value, depending on the user button pressure.

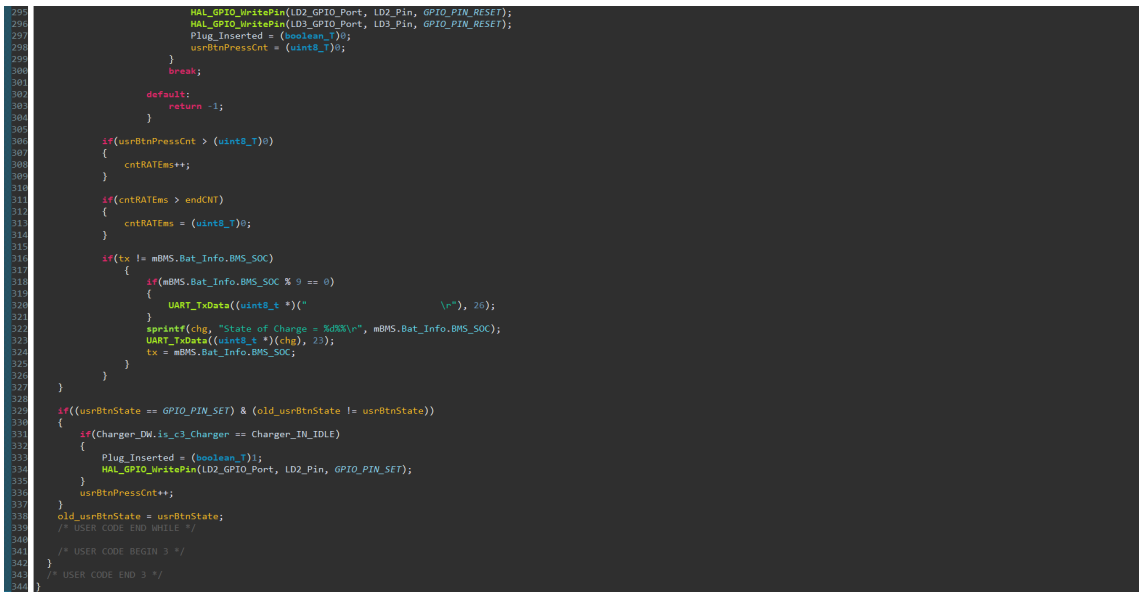


Figure 4.11: User button pressures counter routine

After the closing curly bracket, line 327, of the aforementioned if statement, the program inspects the user button pin state and compares its current value to the old one. Last measure prevents to count multiple times within a single pressure. Meeting these conditions allows to change Plug_Inserted to high value and to switch on the blue LED, stating that plug insertion occurred, only if Charger is in IDLE state. In any case the program increments *usrBtnPressCnt*, i.e. the counter that keeps track of the number of pressures, and it updates the old value of user button pin state, once out of the external if construct.
The counter *usrBtnPressCnt* is zeroed whenever a transition requiring the user button pressure occurs. For instance at the top part of Figure 4.11, at line 298 the program executes a reset of the counter, along with the preceding instructions. The sum of the last ones with the reset makes up the contents of an if statement, which in turn is into the case construct active when Charger is in STOP. While in this state a pressure detection makes the program move forward to switch off the blue and red LEDs, bring down Plug_Inserted and reset *usrBtnPressCnt*. The break instructs the program to exit from the switch-case construct.
An increment to a different counter, namely *cntRATEms*, follows as a consequence of user button pressure, precisely of its counter update therefore bigger than one. This counter makes up for setting a time window in which multiple pressures are taken into account. Once reaching the upper limit *endCNT* the counter is reset.

The transmission of the SOC level over the UART follows. It runs only as an update, in other words the same SOC value is not sent twice in a row thus a transmission takes places in case the next value is different from the old one. An example is provided at the top right of Figure 4.10.
Back to line 156 the program executes the switch command evaluating the value of Charger_DW struct member, called *position* in the previous analysis, which tracks which state, that is not a child one, Charger is into. The inputs management is coded following the same fashion that Embedder Coder opts for converting the state diagram to code. The required input by Charger to proceed through the control algorithm, i.e. the state diagram, is sent automatically in accordance with the conditions to be met. The latter depend on the transition.
A LEDs management routine has been added to ease the debugging phase and monitor the charging process. In particular the green LED states a charging activity when blinking and its completion when still. The blue LED is on when plug is inserted, otherwise off. The red LED blinks when a manual stop operation is being performed. It blinks faster during SOC recalibration procedure and it is on when Charger is in STOP, after the reception of the command to unlock the plug.
When Charger stays in CHARGING_PROCESS a linear increment of the maximum battery cell tension, whose value is used to calculate the SOC which consequently follows the same trend, has been modeled. A similar behaviour is opted for the discharge, albeit with a flatter and clearly negative slope. A discharging process occurs whenever Charger is in standby or stop. Namely while it awaits the command issued by the user button pressure in STOP, MANUAL_STOP and FULL_CHARGE.
For the last case Figure 4.10 provides the discharging process implementation from line 201 to 212. The counter *tmpCNT* sets a reduction rate twenty times smaller than the increasing one. The decrease takes place if the charge is bigger than zero, thereupon the counter is reset and the tension is calculated. The out of charge circumstance is tackled separately in the ensuing else if statement. The SOC value is derived by the tension. Both are assigned to their relative BMS signal responsible to update the Charger.
The charging routine is implemented at lines 194-195-196 and occurs if the conditions guarding transitions with higher priority are not met. Indeed the first evaluated condition is to perform the SOC recalibration. The check on charging process completion follows and finally a verification on the user button pressures counter that if fulfilled leads Charger to carry out a manual stop operation.
To complete Figure 4.11 analysis starting at line 214 an example on how the branching out of the code has been handled. Within a time interval set by the values of *cntRATEms* and *endCNT*, *usrBtnPressCnt* is updated. Once the first counter equals the second, depending on the number of user button pressures a proper value is assigned to the related input. The final LOCs deal with the two counters reset and an increment of *cntRATEms*, which will be zeroed at a subsequent line after the switch construct.

Chapter 5

Conclusion

The initial goal of the thesis differed from the achieved one. The activity aimed at the modeling, verification and validation of an On Board Charger algorithm according to MIL and SIL technologies. The code integration and charging routine development into a STM board has been considered at a later stage. While setting the activity objective the Model Based Design training had to be taken into account, as this topic has not previously been addressed during the course of university.
This document describes the internship project carried out on behalf of Teoresi S.p.A. and it starts with the system requirements reception furnished by the latter and a system overview. It proceeds with their conversion into software requisites, accounting for signals representation into non-virtual bus objects and including their characterization.
Once the signal framework is complete the next step hinges on the charging process design. For this phase Stateflow was used as the main tool to develop the Charger logic along with the Vehicle Control Unit logic opting for finite state machines. The strategy consisted of facing the modeling in two processes, namely the Inlet lock management and the SOC recalibration. The latter development is contemporaneous to its conjunction with the first one, leading to the final model. Both have been initially analysed, tackling them through a further division, thereupon a solution has been implemented and undergone a testing procedure. The MIL has been performed explaining the sequence of operations making it up. The realised test harness for each model, together with simulations, and the coverage analysis are provided including a strategy to maximise the latter.
The SIL is addressed only to the final model and performed employing Simulink Test Manager, as for the MIL, through a simulation test and an equivalence one which compares its outcome with the MIL result. A strategy to equal the SIL coverage to the MIL one is further supplied. Evidences of the equivalence criteria fulfillment are supplied, proving the code behaves as the model. Checks on the ISO 26262 have been run on the final model before the testing phase, followed by the adopted measures to conform the model to the safety standards.
The last chapter is dedicated to the code generation and integration. An operations sequence is mandatory on the model settings to prepare it to generate code from it and some measures are presented to customize the process and the outcome. An examination of the generated files content follows pointing out the variables characterization, such as type and name which are consequences of the customization, the code structure and the model entry functions, successively employed in the imple-

mented charging routine on the board microcontroller in the last stage.
The final section indeed illustrates the code integration and the aforementioned handwritten code. Starting with presenting the target hardware, i.e. STM32F439ZI Nucleo-144 including its features, the aim of the routine has been thoroughly explained. After an introduction to the IDE and the device setup procedure, the code structure has been analysed. Moreover the LEDs routine was explained, to be able to track the charging progress, together with the user button management to interact with the algorithm.

Bibliography

[1] S. Popinchalk. "Nonvirtual bus signals." (Apr. 2008), [Online]. Available: https://blogs.mathworks.com/simulink/2008/04/29/nonvirtual-bus-signals/.

[2] K. Ghani and J. A. Clark, Eds., *Automatic Test Data Generation for Multiple Condition and MCDC Coverage*, Control Flow Coverage Criteria, 2009, ed. by IEEE.

[3] C. Morley, "Automotive industry life cycle analysis: Shorter timelines," 2019. [Online]. Available: https://www.jabil.com/blog/automotive-industry-trends-point-to-shorter-product-development-cycles.html.

[4] MathWorks. "Introduzione allo standard ISO 26262 e al workflow Model-Based per gli standard di sicurezza." (Jul. 2020), [Online]. Available: https://it.mathworks.com/videos/introduction-to-iso-26262-and-model-based-design-workflow-for-safety-standards-1595422067098.html?s_tid=srchtitle_model%5C%20based%5C%20workflow_1.

[5] ——, "Model-based design with matlab and simulink." (Apr. 2020), [Online]. Available: https://it.mathworks.com/videos/model-based-design-with-matlab-and-simulink-69040.html.

[6] STMicroelectronics, *UM1974 User manual STM32 Nucleo-144 boards*, Aug. 2020, ch. Features. [Online]. Available: https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo-144-boards-stmicroelectronics.pdf.

[7] MathWorks, *Stateflow getting started guide*, Sep. 2021, ch. Stateflow Fundamentals, pp. 2–2. [Online]. Available: https://it.mathworks.com/help/pdf_doc/stateflow/stateflow_gs.pdf.

[8] ——, "Test harness construction for specific model elements." (2022), [Online]. Available: https://it.mathworks.com/help/sltest/ug/test-harness-construction-for-specific-model-elements.html.

[9] ——, "Manage build process files, rtwtypes.h." (), [Online]. Available: https://it.mathworks.com/help/rtw/ug/build-process-files.html#f1159207.

[10] ——, "Manage file packaging of generated code modules, Generated code modules." (), [Online]. Available: https://it.mathworks.com/help/ecoder/ug/generate-code-modules.html.

[11] ——, "Model reference basics." (), [Online]. Available: https://it.mathworks.com/help/simulink/ug/overview-of-model-referencing-1.html.

[12] ——, "TLC files." (), [Online]. Available: https://it.mathworks.com/help/releases/R2020b/rtw/tlc/tlc-files.html.