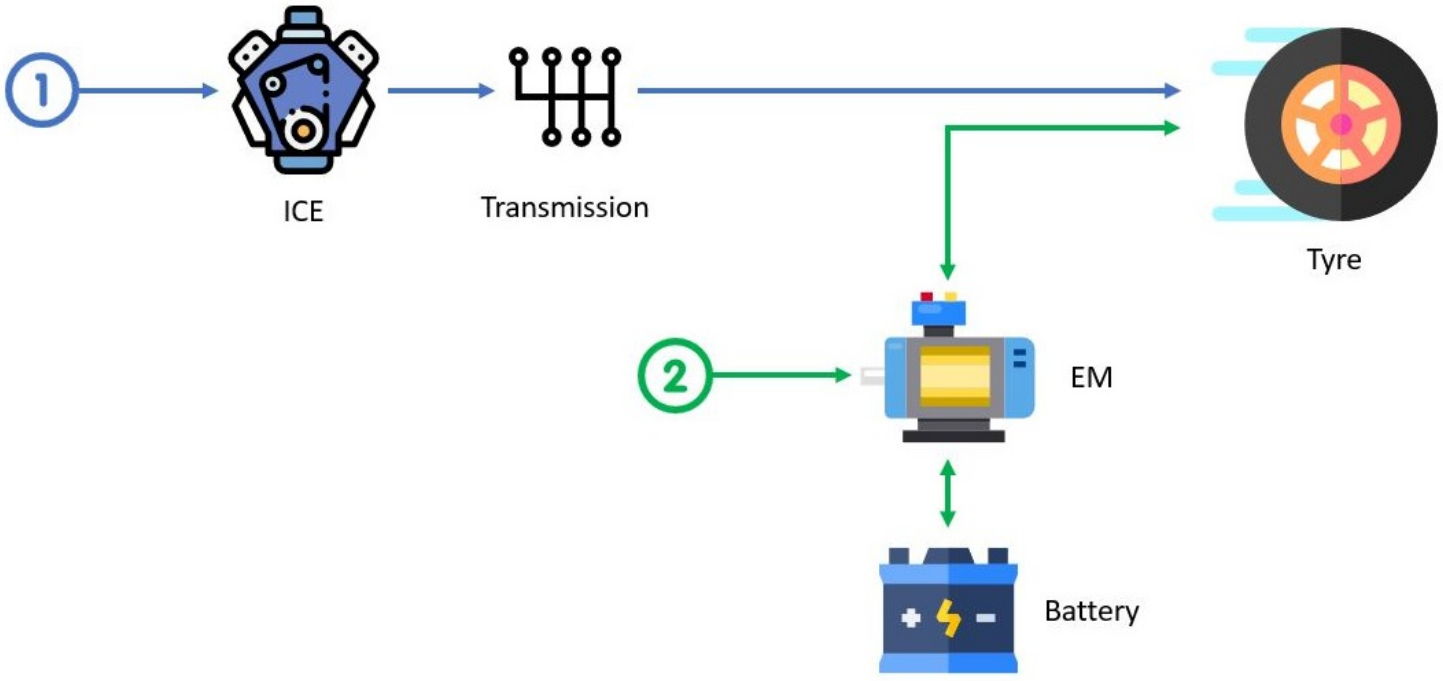
Controller design for a PHEV

Gallina, Gargano, Mirabella
Prignoli, Rubbini

Compliance Design of Automotive Systems
A.Y. 2020/2021



Controller design for a PHEV

Gallina, Gargano, Mirabella, Prignoli, Rubbini

MOTORVEHICLE
UNIVERSITY OF
EMILIA-ROMAGNA

Preface

In this report, we show how Model-Based Design can be applied in the development of a controller for a hybrid electric vehicle. To do so we had to previously define the environment in which the controller will operate starting with the design requirements. They can be found in the development of component models of the physical system, such as the power distribution system and mechanical driveline. We also show the development of an energy management strategy for several modes of operation including the full electric, hybrid, and full combustion engine modes. All the presented steps, together with the Code generation and Testing, have been implemented and simulated in Matlab/Simulink environment.

Gallina, Gargano, Mirabella, Prignoli, Rubbini
February 2021

Contents

List of Figures

List of Tables

1

Introduction

The reason for the choice of a hybrid vehicle related topic is because we all believe that it will be one of the main challenges for the future of automotive in next years, with still a lot of improvements in several fields, like those available in the control strategy of the current PHEVs & MHEVs, especially with the future development of smart cities and V2X solutions. In this report, we explore key aspects of hybrid electric vehicle design and outline how Model-Based Design can offer an efficient solution to some of the key issues related to this topic. Due to the limited scope of the report, we do not expect to solve the problem in totality or offer an optimal design solution. Instead, we offer an example that will illustrate the potential benefits of using Model-Based Design in the engineering workflow. Traditionally, Model-Based Design has been used primarily for controller development. One of the goals of this report is to show how this approach can be used throughout the entire system design process.

- In *Section 2* we offer a short overview of PHEVs, highlighting the various aspects of the design with a focus on the used architecture.
- *Section 3* is used to exploit Model-Based Design features and how we decide to apply them to the PHEV controller development.
- With *Section 4* we will go through the Simulation and Testing phase providing a complete description of all the steps done to validate our controller. Starting from the Unit testing, we continue with the MIL simulation and, after having provided an Automatic code generation thanks to the Simulink C coder plug-in, we will conclude with the SIL simulation.
- Finally, in *Section 5* we conclude our work and discuss possible future developments on it.

2

Plug-in Hybrid Vehicle design

This section describes the design considerations of the system and is also a reference for all the key parameters of PHEV system components.

2.1. Hybrid architecture and Energy management

2.1.1. Hybrid architecture

A hybrid electric vehicle has both features of an electric vehicle and an *Internal Combustion Engine* (*ICE*) vehicle: at low speeds, it operates as an electric vehicle with the battery supplying the *Electric Motor* (*EM*). At higher speeds, the engine and the battery work together to meet the drive power demand.

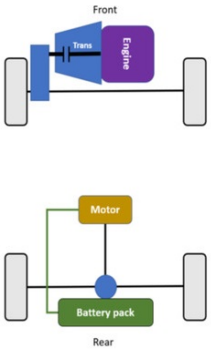


Figure 2.1: P4 parallel architecture

The starting design choice is a **Parallel Hybrid architecture**, where ICE is mechanically connected to the wheels and can therefore directly supply mechanical power to them. The EM is added to the drive train in parallel to the ICE, so that it can supplement the ICE torque. Regenerative braking is considered as well. This type of architecture has been chosen among all the others available since it is the one that better suits a Plug-in hybrid vehicle. Here in fact the electric motor is not just helping the ICE but, under specific circumstances, can also be able to supply power to the car, even for several kilometers, until the battery pack discharges. In order to simplify the model of the entire system, a **P4 configuration** (Figure 2.1) has been chosen. In this peculiar configuration, the ICE supplies power to the front axle through the transmission, whilst the motor generator unit is connected to the rear axle only.

2.1.2. Power flows

The difference between using the ICE and the EM to drive the wheels is explained in Figure 2.2.

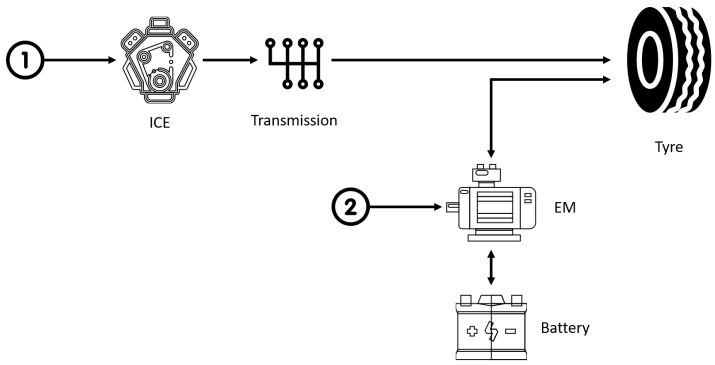


Figure 2.2: Power flows

The system comprises two different energy paths, that can be combined when required:

1. In Path 1 ICE is used. Here the energy flows directly from the engine through the transmission to the wheels.
2. EM energy flow is in Path 2. In this case, the energy flows from the battery to the EM so that it can drive wheels (EM as a motor). Notice that energy can flow also in the reverse direction to store the energy coming from the Regenerative braking system (EM as a generator).

2.2. Design considerations

This part of the report is dedicated to a brief discussion of design considerations and the key aspects of the component design.

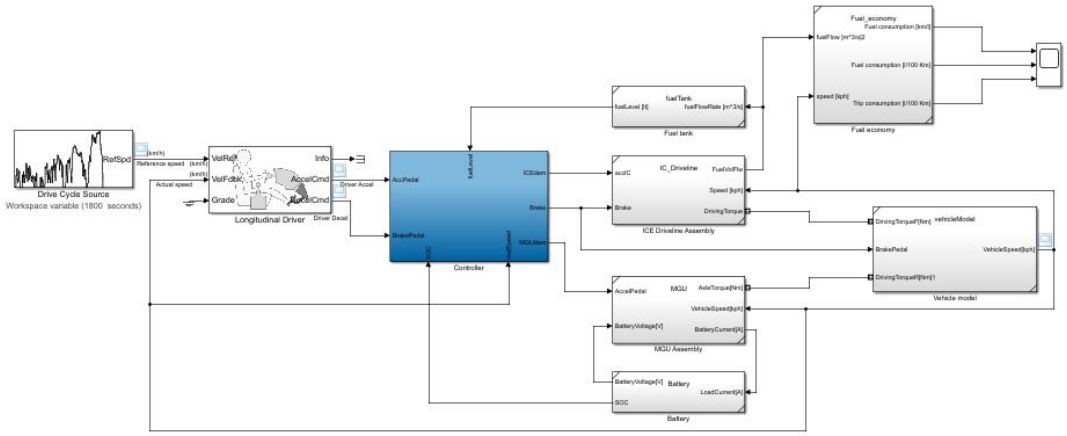


Figure 2.3: Full System

Starting from the left in Figure 2.3, there is the *WLTP cycle* block, used as driving cycle to test the system. The **Worldwide harmonized Light vehicles Test Procedure** (**WLTP**) cycle goes through four different phases: low, medium, high, and extra high speed. It is suitable to test the control strategy in order to verify the correct behavior of the controller. It feeds, together with the actual car speed, the *Longitudinal Driver* block, provided by Simulink, that will emulate a driver giving as output the Acceleration or Deceleration demand. The *Longitudinal driver* block simply implements a PI controller that works on the speed error with respect to the reference one.

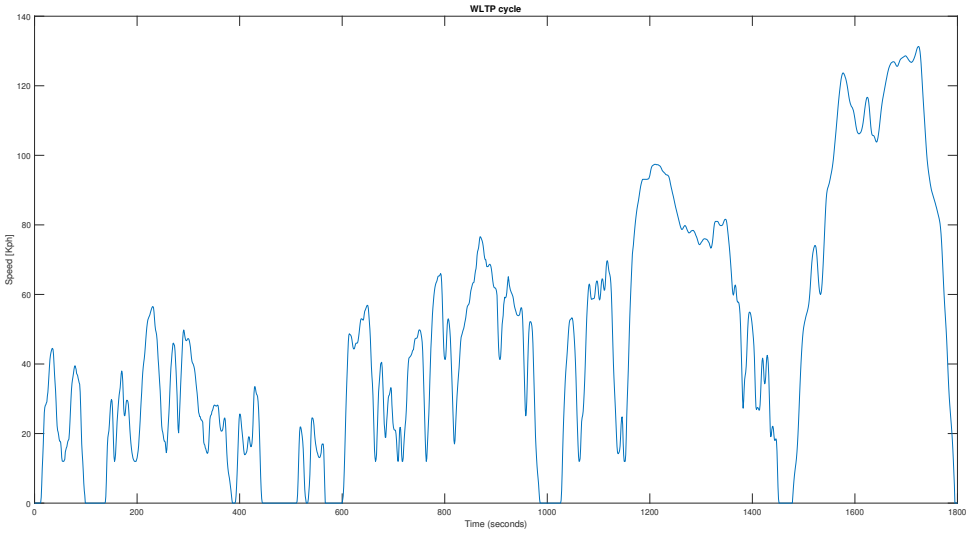


Figure 2.4: WLTP class 3b drive cycle

2.2.1. Internal Combustion Engine

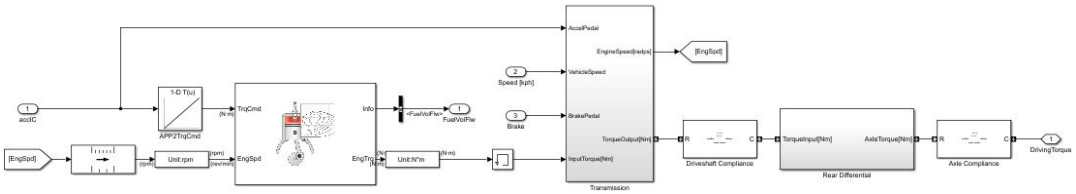


Figure 2.5: ICE driveline

The engine is modeled with a *Mapped Spark-Ignition* (*SI*) engine block, recommended for different kinds of simulations.

Input	Output
• Acceleration request • Brake intensity • Actual speed in kph	• Fuel volume flow • Driving torque

Table 2.1: ICE I/O

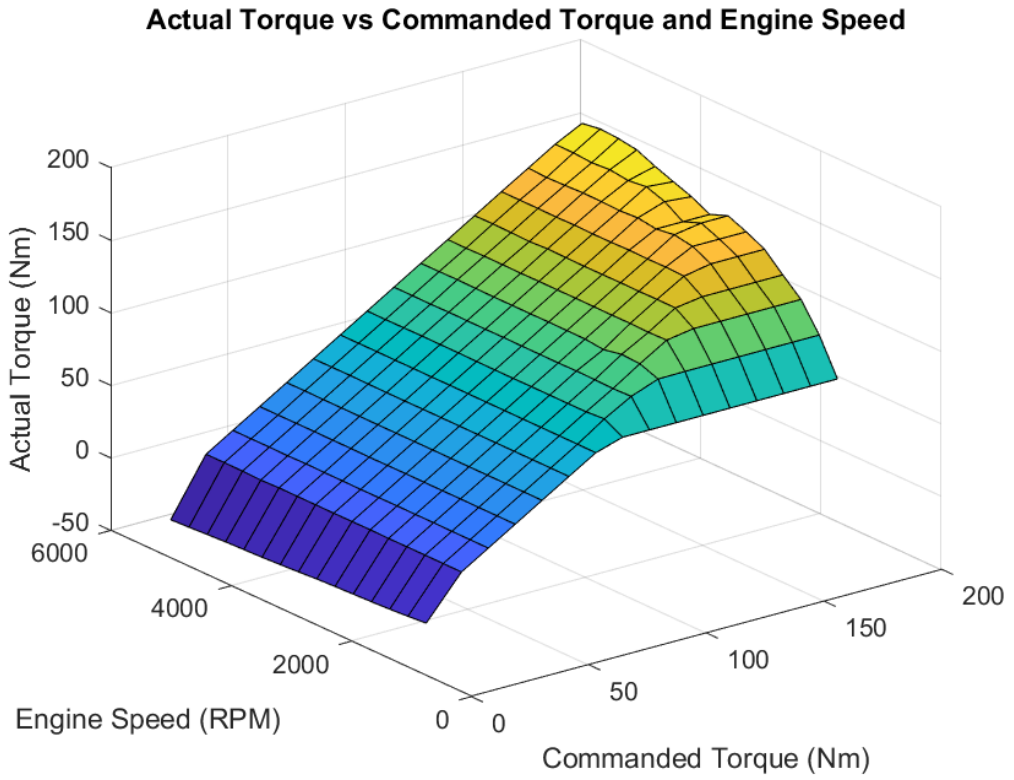


Figure 2.6: IC Mapped Torque

This driveline unit is provided by MathWorks and so there is no need to validate it in this project. It allows taking as input just Acceleration and Braking, provided by the controller, and the actual speed as system feedback, to give back in output the driving torque generated by the ICE and the fuel volume flow that can be used to analyze the fuel consumption and subsequently the efficiency of the vehicle.

2.2.2. Motor Generator Unit

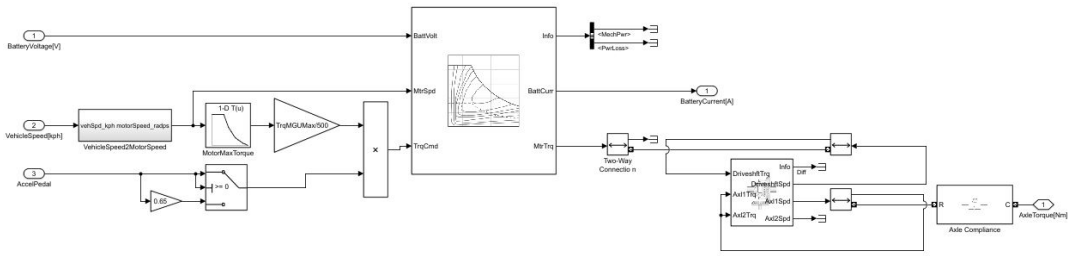


Figure 2.7: MGU driveline

The *Motor Generator Unit* (*MGU*) is modeled with a *Mapped motor* block, which allows to apply a torque-based control.

Input	Output
• Acceleration request • Battery voltage • Actual speed in kph	• Battery current • Driving torque

Table 2.2: MGU I/O

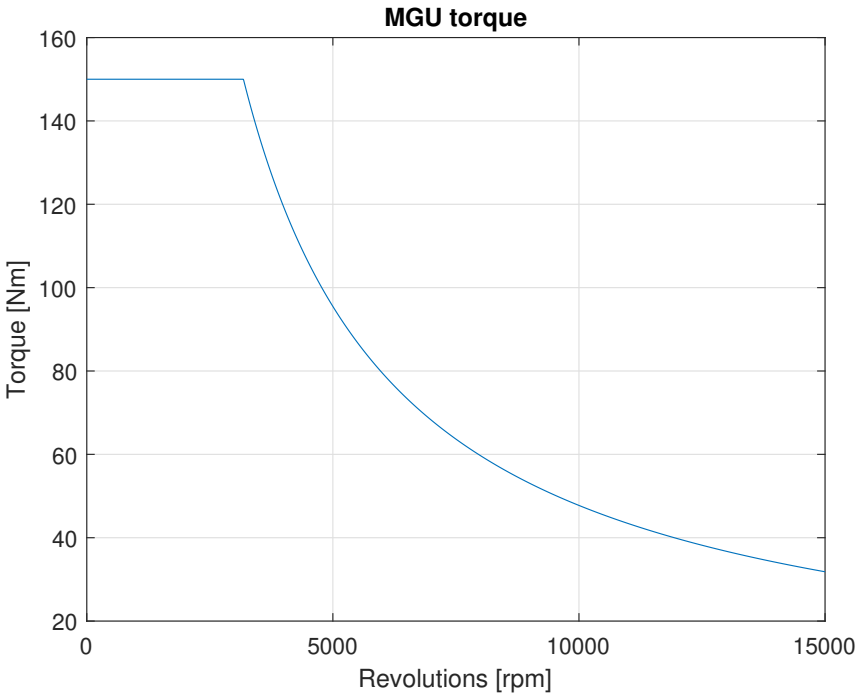


Figure 2.8: MGU torque

This module has been taken from a known authority (MathWorks) as well and so there is no need to validate it. It requires just 3 inputs to work: the Acceleration request that comes from the controller, the Actual speed provided as system feedback and the Battery voltage coming from the battery system that will be described in the following subsection. As outputs there are again the driving torque and the battery current, needed to feed the battery pack module. To properly model the regen efficiency of the system, a multiplication constant of 0.65 has been added when the requested torque is negative (i.e. generator mode).

2.2.3. Battery Pack

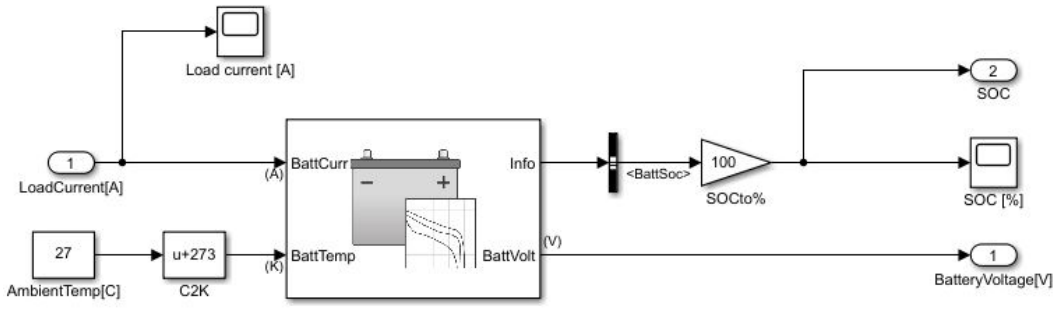


Figure 2.9: Battery module

The main considerations in the battery design are referred to its capacity and discharge characteristics.

Input	Output
• Load current	• SOC • Battery voltage

Table 2.3: Battery Pack I/O

This block is strictly connected to the MGU driveline one. It takes as only input the load current generated as output from Figure 2.7 and provides as output the battery voltage that feeds the MGU and the *State of Charge* (*SOC*) as system indicator, needed to correctly perform the control strategy.

2.2.4. Transmission and Differential

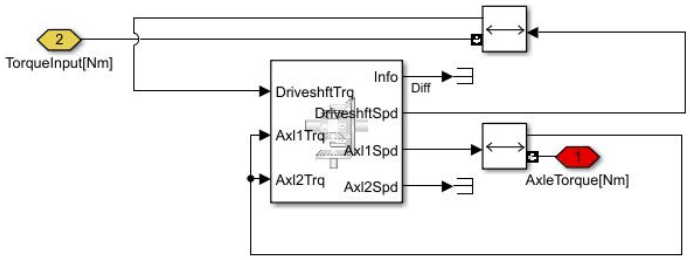


Figure 2.10: Transmission and Differential Model

The transmission system is kept simple as well. It is modeled with an *Ideal fixed gear transmission* block which implements an idealized fixed-gear transmission without a clutch or synchronization. The differential is modeled using an *Open differential* block which implements a planetary bevel gear train.

The transmission features a gearbox driven by a finite state machine that sets the correct gear based on the vehicle speed.

The block has been inserted directly in the ICE and MGU drive lines to allow a clear implementation and easier collection of the various block's inputs and outputs.

2.2.5. Vehicle

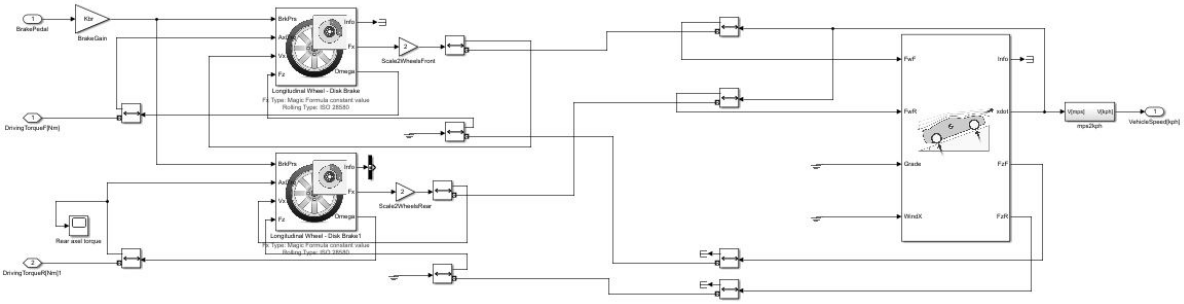


Figure 2.11: Vehicle Model

For the study of vehicle dynamics, the main focus is on its longitudinal dynamics.

Input	Output
• Driving torque front axle (ICE) • Driving torque rear axle (MGU) • Brake pedal	• Vehicle speed

Table 2.4: Vehicle model I/O

This model is positioned on the extreme right of our system. It has the task to take the torque coming from both ICE and MGU and the brake pedal intensity to provide the Vehicle speed. This model implements the wheel dynamics, and so the friction coming from the interaction between tyre and ground (using the Magic Formula) and the longitudinal dynamics of the vehicle (the aerodynamic drag). Moreover, the rolling resistance of the tyres has been implemented according to the *ISO28508* standard. Also this block comes from a trusted authority (Mathworks) and so the validation of this

module is not present in this report. The configurable parameters were chosen with the goal to maximize the vehicle balance and thus its performance.

3

Model Based Design of a PHEV controller

Model-Based Design (**MBD**) provides an efficient approach for establishing a common framework for communication throughout the design process while supporting the development cycle (V-model). In model-based design of control systems, development is manifested in these four steps:

- Modelling a plant,
- Analysing and synthesizing a controller for the plant,
- Simulating the plant and controller,
- Integrating all these phases by deploying the controller.

3.1. Motivation

There are several reasons to prefer MBD over a traditional design methodology. First, MBD allows to save time and reduce errors: indeed, rather than using complex structures or even expensive prototypes of the system, MBD allows to realize plant models with advanced functional characteristics, conceptualizing each of them for a better understanding of the overall system behaviour, particularly important in systems with high multi-domain complexity like the one under analysis.
Moreover, software and hardware implementation requirements are considered together to automatically generate code for embedded deployment. The models that are built in this way are used with simulation tools thus achieving rapid prototyping, software testing, and verification.
The two main modelling approaches are ”data-driven” and ”first principles” based. If from one side, data-driven approach provides typically an easier way to implement models even though limited by data collection capability, including also problems of data over-fitting, on the contrary the first principles approach, following physical laws and implementation through differential equations, is much more flexible. As always, mixing the two methods provides better results.

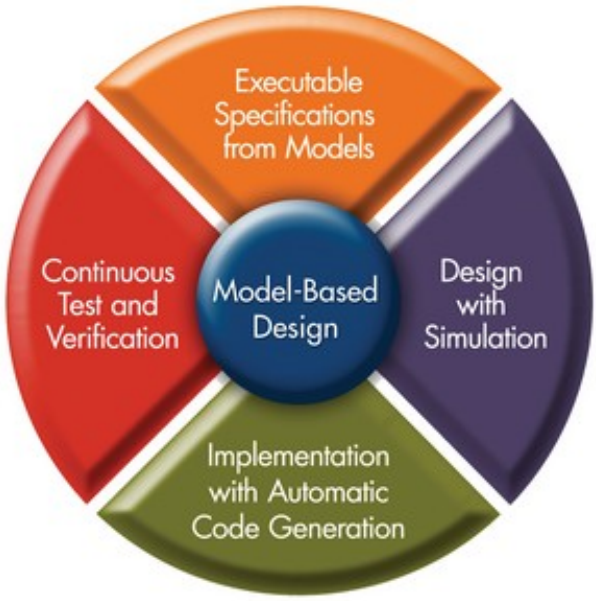


Figure 3.1: Model Based Design

3.2. Requirements

The idea behind the used control approach is to create something that could be easily improved in the future with an increased modularity of the controller. For this reason, the design choice for the controller is to have a cascade of two units:

1. **Vehicle state attribution logic.** The first unit defines the current vehicle state and it is implemented with a Finite State Machine.
2. **Torque Allocation management.** The second unit of the controller is responsible for the torque allocation.

The cascade of these two units gives as output of the controller the demanded torque to both ICE and MGU (as well as the breaking request).

The value added by the resulting structure is modularity, which is a key factor for Unit testing. Moreover, the implementation of the controller is expected to reduce the fuel consumption over the cycle.

3.3. Vehicle State FSM

The natural choice to implement the Vehicle state attribution logic is a *Finite State Machine* (*FSM*). The FSM takes as input the SOC, the Fuel level, the vehicle actual speed and the difference between Acceleration and Braking pedal and provides as output an integer corresponding to one of the 5 states below:

- **Dead (s0)**: this is a safe state that occurs when there is no battery charge and the fuel tank is empty. Here, neither engine nor brakes are working (zero output) even if the driver is asking for an acceleration
- **No Charge (s1)**: when the battery SOC is below the minimum level, only ICE drive is available, thus preventing battery depletion (and eventual damages to the battery). In this case, the maximum torque that the entire system can generate is equal to the ICE maximum torque,
- **Electric Drive (s2)**: When the fuel level is below the minimum threshold, only the MGU drive is available and the maximum torque that the entire system can generate is equal to the EM maximum torque. On other occasions, the vehicle is run merely by the MGU only if the speed is below a defined threshold ($45Kph$),

- **Combined (s3)**: both engines are used simultaneously. In this state, torque partitioning follows this rule: the electrical motor, whose maximum torque is lower than the one provided by the IC engine, starts as soon as there is a request from the driver. The request of the IC engine starts with a delay but in each instant the sum of the two requests satisfies the following relationship:

$$IC_{torque} + MGU_{torque} = Acc_{Pedal} \tag{3.1}$$

  In fact Acc_{Pedal} is normalized with respect to the total torque that the system is able to generate, namely the sum of the two maximum torques,

- **Regenerative Braking (s4)**: The vehicle is in this state every time the driver performs a deceleration. In particular, if the battery is not full, part of the braking torque can be generated by the electrical motor, thus leading to a battery recharge, and part of the torque will be provided by traditional brakes. If the battery is already fully charged, to prevent the break of the battery, the controller sends the total brake request to the brakes.

From	Condition				To
	SOC	Fuel	Acc	Speed	
s0			<0		s4
	>SOCmin		<=accMaxMGU		s2
		>fuelMin	<=accMaxICE		s1
	>SOCmin	>fuelMin	>accMax(MGU,ICE)		s3
s1			<0		s4
	<=SOCmin	<=fuelMin			s0
	>SOCmin			<=speedEDMax	s2
		<=fuelMin			s2
	>SOCmin	>fuelMin	>accMaxICE		s3
s2			<0		s4
	<=SOCmin	<=fuelMin			s0
	<=SOCmin	>fuelMin			s1
	>SOCmin	>fuelMin		>speedEDMax	s3
			>accMaxMGU		s3
s3			<0		s4
	<=SOCmin	<=fuelMin			s0
	<=SOCmin		<=accMaxMGU	<=speedEDMax	s2
	<=SOCmin	>fuelMin			s1
s4			<0		s4
	<=SOCmin	<=fuelMin			s0
	>SOCmin			>speedEDMax	s2
	<=SOCmin	>fuelMin			s1
	>SOCmin	>fuelMin		>speedEDMax	s3

Table 3.1: Control strategy

3.4. Torque Demand Controller

This unit can be easily identified as a black box, having 4 inputs and 3 outputs. The inputs are:

- **State**: this is the outcome of the FSM, and can assume 5 possible values (see Section 3.3).
- **AccPedal**: this is the request coming from the driver. It is a normalized value between 0 (meaning no acceleration is requested) and 1 (maximum acceleration demand).
- **BrakePedal**: this is the braking request, again coming from the driver. Similarly to AccPedal, also BrakePedal is normalized between 0 and 1.
- **SOC**: this input is the battery State of Charge. For this reason, it comes directly from the battery and it is given in percentage from 0% (battery fully depleted) to 100% (fully charged battery).

The outputs are:

- **ICreq**: this signal is the torque request to the IC engine. Its value is between 0 and 1. The maximum load for the IC engine corresponds to ICreq = 1. The normalization is done with respect to the maximum torque provided by the IC engine.
- **MGUreq**: this signal refers to the torque request to the EM. Again, the value is normalized with respect to the maximum torque that can be generated by the MGU. The difference with *ICreq* is that this value can also be negative, so that the regenerative braking can take place during braking operations. The negative values are normalized with respect to the maximum braking torque which can be generated by the MGU.
- **Brake**: this is the braking torque applied to the front and rear wheels. The normalization parameter is the maximum torque generated by brakes.

To obtain the desired behaviour, the system is divided in sub-units, each corresponding to a state. The state is selected through a *Switch* block, taking the state as input. The remaining inputs are propagated inside each of the sub-units as shown in Figure 3.2

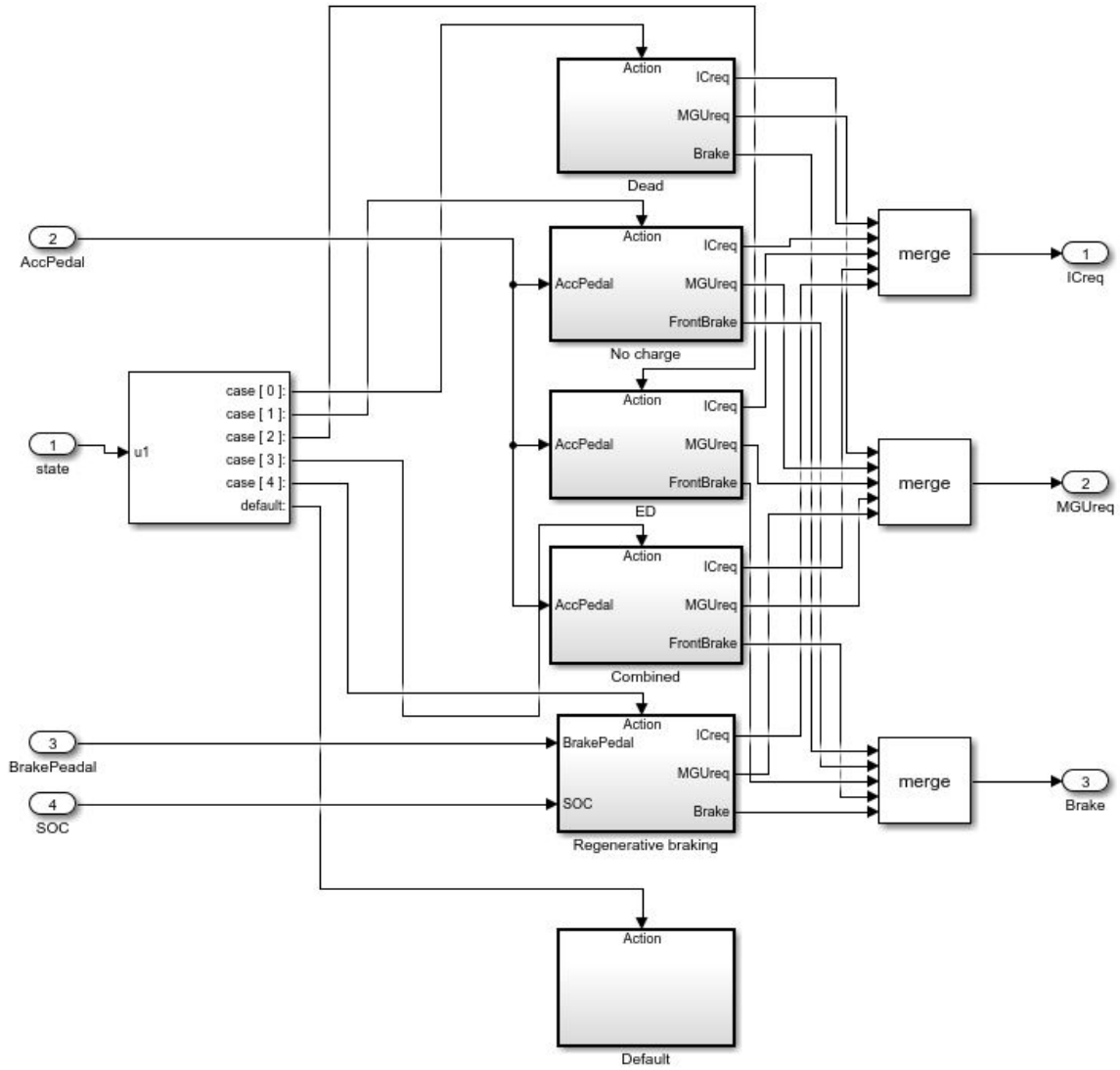


Figure 3.2: Torque-Brake Allocator block scheme

As it can be noticed from Figure 3.2, the *Dead* case is fed with no input, Acc_{Pedal} goes into *No Charge*, *ED*, and *Combined* states whilst $Brake_{Pedal}$ and SOC are inputs just for *Regenerative Braking* sub-unit.

The only two states that have one degree of freedom in the torque partitioning are the *Combined* and *Regenerative braking* states. A simple but effective design choice is to exploit a *Low-Pass Filter* (*LPF*) for smoothing the torque request to the IC engine, preventing an abrupt increment of fuel flow rate and thus drastically reducing fuel consumption. The missing torque demand is fed to the electric motor, which has the big advantage of providing the requested torque with a very fast dynamic and with high efficiency. The resulting behaviour is that, typically, the electric motor works intermittently, whereas the IC engine has a slow dynamic. There are two not ordinary conditions:

- If the electric motor is not able to handle a sudden torque request, the LPF is bypassed so that the IC engine is demanded with no delay.
- If the steady-state torque demand cannot be handled by the IC engine, the electric motor is asked to compensate for the missing torque request.

Similarly, in the *Regenerative braking* state an LP filter is exploited to split the working regions between the disk brake and the regenerative brake, that are respectively during the ”transient phase” and during the ”steady-state phase”.

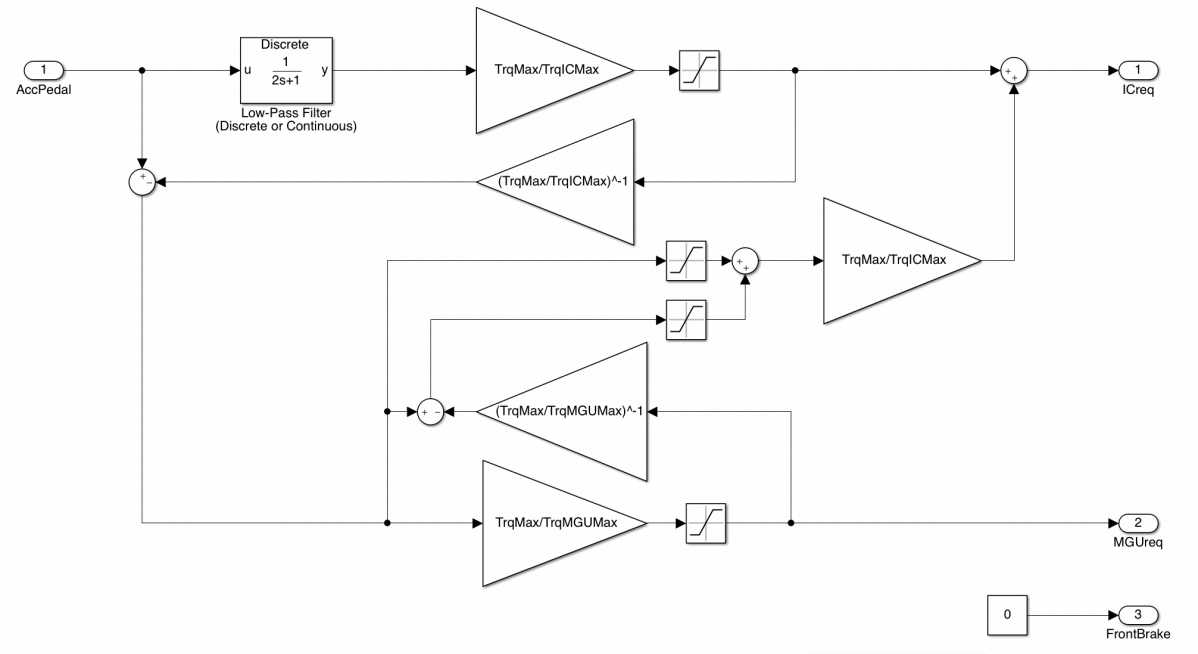


Figure 3.3: Combined state block

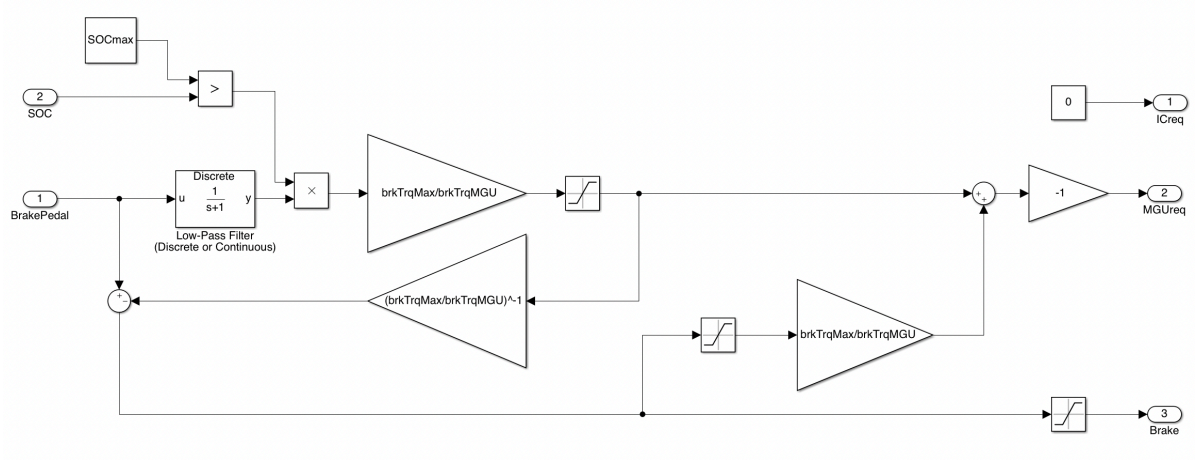


Figure 3.4: Regenerative Braking state block

In the subsystems shown in Figure 3.3 and Figure 3.4 the low pass filters are present in their discrete implementation. The saturation blocks ensure every request stays between 0 and 1. In addition, in Figure 3.4 the initial check on the SOC guarantees the MGU provides zero braking torque to avoid battery overcharge.

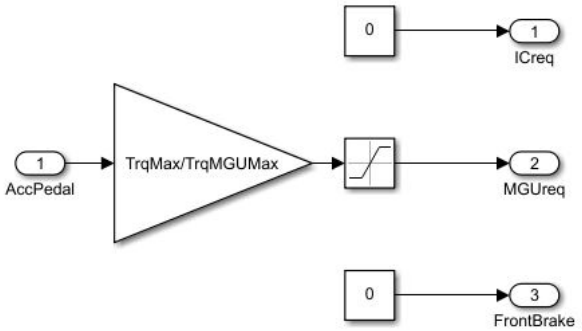


Figure 3.5: Electric Drive state block

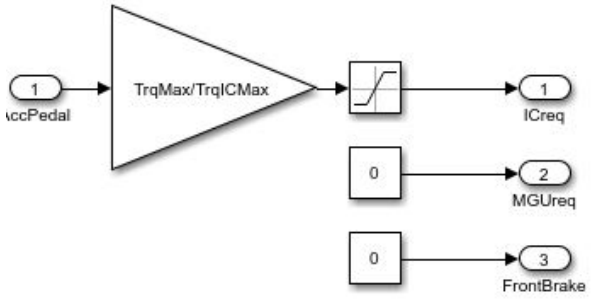


Figure 3.6: No Charge state block

The subsystems in Figure 3.5 and Figure 3.6 show the implementation of the two states where just one between ICE or MGU is working and so the total torque provided to the axle comes directly from one single entity. The saturation blocks are added also here to ensure every request stays between 0 and 1.

Figure 3.7: Dead state block

The subsystem in Figure 3.7 shows the implementation of the safe state. Here all the torque requests are forced to zero by easily use a constant to feed all the output of the controller.

4

Simulation and testing

4.1. Unit testing

4.1.1. Finite State Machine Testing

The purpose of testing the Finite State Machine is:

- Verify that all the transitions from one state to another are thought correctly
- Parameters are well defined and included in the right work-space so that they can be used by the state machine
- The sampling time is set correctly and does not produce weird behaviors.

To create the test bench for the FSM, it has been used *Simulink Test*, through the *Test Manager* interface. Here different test cases have been created trying to cover all the possible arrows highlighting a transition between two states. For this purpose, the *Baseline Test* provided by Test Manager has been quite handy.
First of all, once the test suite has been created it is necessary to define all the test scenarios; according to the following information, some considerations can be done:

- **5 Different states**: *Dead*, *No Charge*, *Electrical Drive*, *Combined* and *Regenerative Braking*.
- Every state considers a transition to another state or to itself; therefore, for every state there are 5 transitions to be checked.
- **Workspace parameters**: maximum driving and braking torques for IC engine and MGU, battery parameters, fuel information, maximum speed in *Electrical Drive*
- **5 Inputs**: *realSpeed*, *Fuel*, *SOC*, *AccPedal* and *BrakePedal*.

Firstly, 25 different test cases have been defined to check all transitions. The different scenarios have been modeled by means of **Signal Editor** facility which helps in creating signals feeding the input of the FSM. The signal file can be saved in the Matlab workspace or as a *.mat* file and then passed in the input section of the test case. In Figure 4.1 it is shown an example of scenario creation, in that case to test transition between *Electrical Drive* to *Regenerative Braking* state. It can be seen that, after 5 seconds the Acc_{Pedal} signal becomes zero whereas the $Brake_{Pedal}$ signal suddenly rises, so that to trigger the state transition.
Every signal can be created in three different ways: by drawing it, by filling the table below it (as shown in 4.1) or by importing data (as arrays) from the workspace.
The signal creation procedure through **Signal Editor** has been repeated for every transition check, leading to the creation of 5 different test suites each one containing 5 different scenarios.

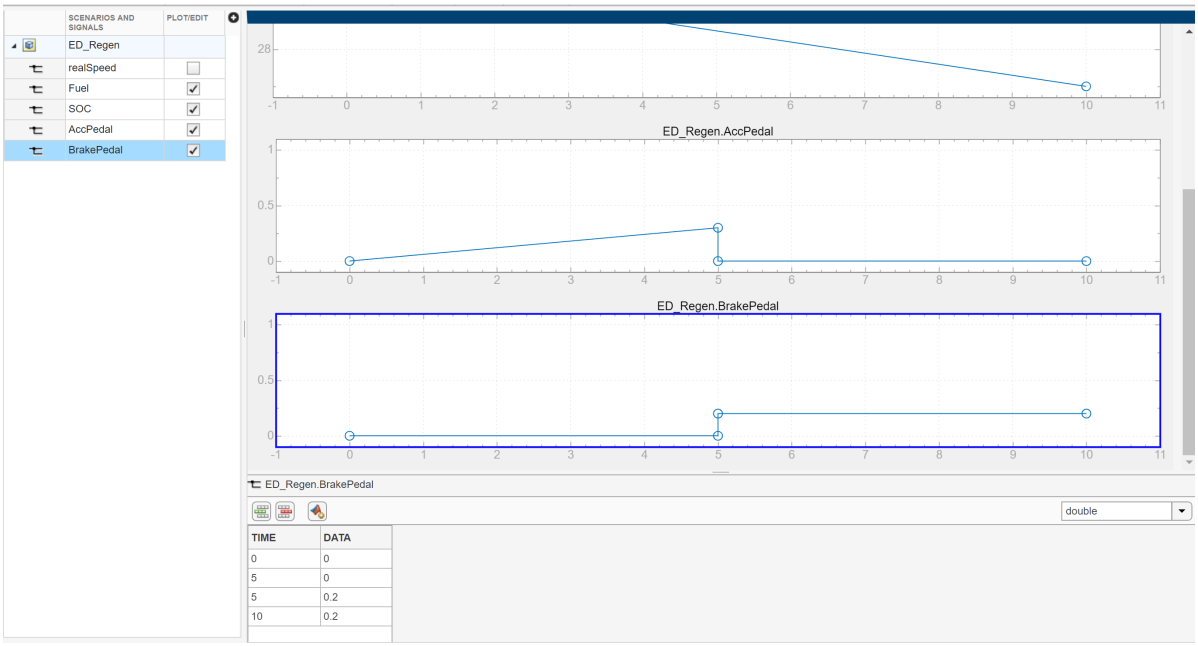


Figure 4.1: Signal Editor

Once the inputs have been mapped to the ones of the FSM, some baselines can be created to check whether the simulation results, in terms of monitored signals, match the expected behaviours. For this purpose, every test case is completed with the introduction of baseline signals, again created through **Signal Editor**. An important detail is that if these signals have the same name of the FSM output, that is *'state'*, then it will not be necessary to specify the simulation outputs, because this information is already included in the model passed to the test bench (that is the Simulink model of the FSM).
Together with the baseline, some tolerances can be defined in order to consider the result acceptable when the difference falls in the range within the tolerance. Possible settings are:

- Absolute tolerance: related to the absolute difference between baseline and output signals.
- Relative tolerance: related to the ratio of the difference between baseline and output with respect to the baseline itself.
- Leading tolerance: used for the signal whose change occurs ahead of the baseline signal.
- Lagging tolerance: used for the signal whose change occurs after the baseline signal.

It is worth underlying that the first two tolerances shall consider mostly the type of solver whilst the choice of the last two depends on the simulation step size and the sampling time of the FSM.
Before starting the test, *Simulink Test* allows to create the *Test Specification Report* by means of an automatic report generation.

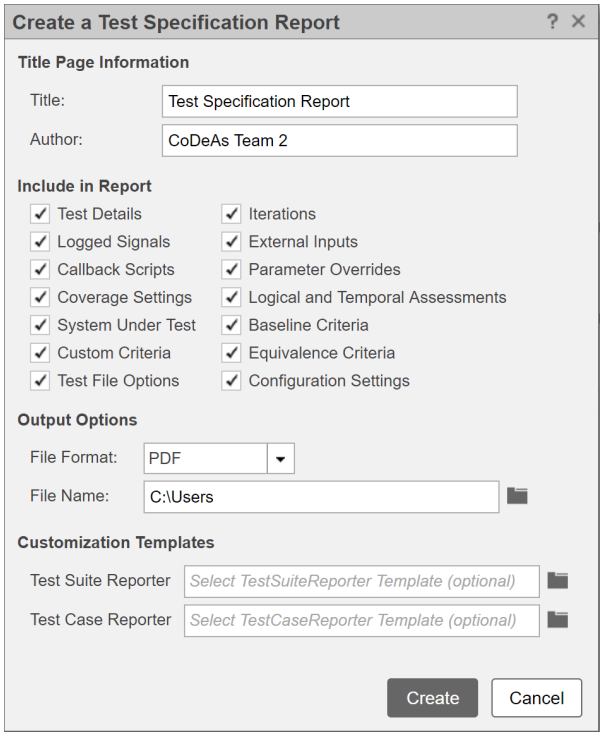


Figure 4.2: Test Specification Report creation

In the window panel shown in Figure 4.2 a lot of options can be flagged, so that additional information is included in the report.
After that, the test can be carried, and the results will appear in the result window as shown in 4.3. The results are given in a simple expandable tree, accompanied by a green check if the baseline test went successfully. By exploring it, all the single test cases can be analysed also by means of visual plots, plots relative to previously set tolerances and plots of comparison between baseline and output of each simulation.

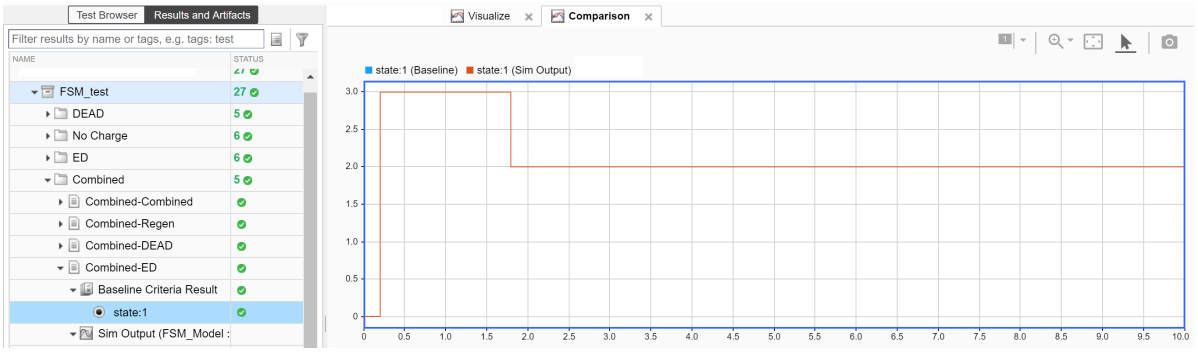


Figure 4.3: Test Results for the FSM

Now that the test is completed it is also possible to automatically generate the result report by clicking on the specific button of **Test Manager**.

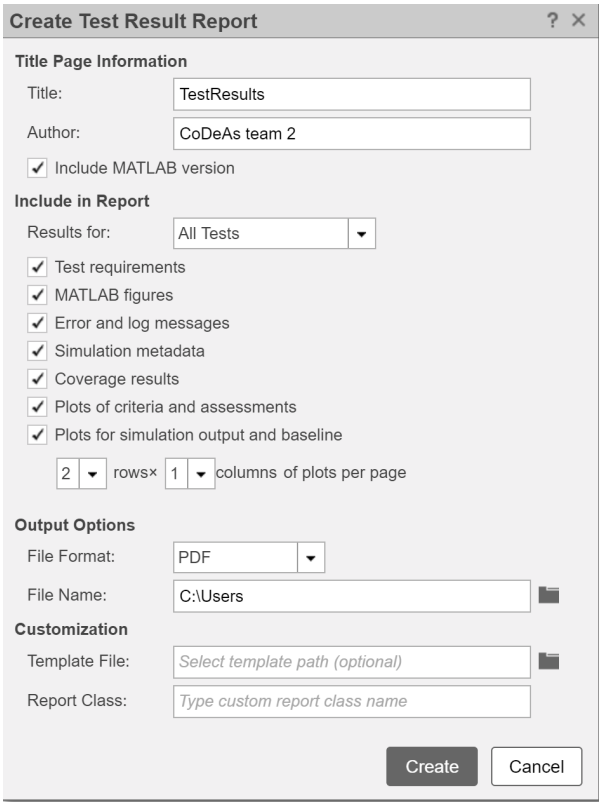


Figure 4.4: Test Result Report creation

Again, as it should be easily seen from Figure 4.4, a lot of information can be included in the report, even the MATLAB figures. The report format can be decided between PDF, DOCX and ZIP.

4.1.2. Torque Demand Controller Testing

In this subsection are discussed the test related to the *Torque Demand Controller*. Firstly, it is necessary to describe how the test bench has been built, starting from the requirements of the unit under test. The main requirement of the unit is providing the right commands as output, according to:

- **Current state**: every state is treated independently from all the others. In this way it becomes easier to define the set of inputs, limiting them to those expected for the specific state.
- **Electrical motor characteristics**: the MGU can provide a maximum torque, specified as a workspace parameter. It is also capable of serving the regenerative braking, providing a negative torque. Also, this last torque has a limit that the controller has to know and respect.
- **Internal combustion engine characteristics**: the IC can provide a maximum torque, specified as a workspace parameter.
- **Wheel brakes**: the brakes have a limit on the maximum applicable torque.

The test bench has been created using the tool Simulink Test. With Simulink Test it is possible to create non-intrusive test harnesses to isolate the component under test. In addition, it allows the definition of requirements-based assessments using a text-based language, and the specification of test input, expected outputs, and tolerances in a variety of formats.
As a primary step, the controller subsystem has been turned into a Simulink model, so that it can be passed to the Test Manager. It follows the creation of a test file, for the controller, including test suites for each of the 5 states of the system.
Every suite has been named with the state name and filled with possible test cases or scenarios.
Indeed, once inside a test suite, the tool allows the creation of test cases. Among the type of tests it is worth mentioning:

- **Baseline Test**: one of the most used for this unit. The test simulates the controller, takes some outputs or intermediate signals and compares them with a predefined baseline representing the desired behaviour. If everything matches or stays within well-defined tolerances, the test is passed.
- **Equivalence Test**: compares the output of two simulations. It has not been used during this test phase.
- **Simulation test**: performs a simulation with no criteria, unless specified. It is useful when building the test bench itself and can be used also to get reference signals that should then be exploited as ground truth during SIL tests.

To sum up, there are 5 test suites, one for each state and every suite contains one or more cases, depending on the number of checks that need to be performed.
In the single test case, the Test Manager allows to pass the input signals, which need to be mapped with the ones of the model. To perform this step, an automatic function can be executed through the button **Map Inputs**. The inputs defined into the *.mat* or *.xlc* file will be automatically linked to the inputs of the model, provided that they share the same identifier or name.

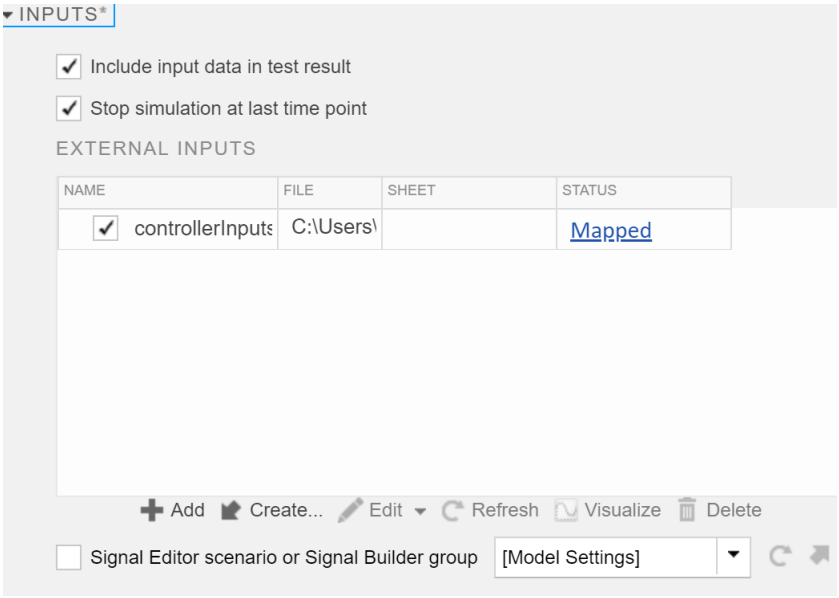


Figure 4.5: Input windows of Test Manager

There is a utility to create proper inputs called **Signal Editor**, through which signals can be drawn or be described by tables (like in *Microsoft Excel*) and the table can also be generated starting from a workspace variable (likely an array of values).
The output can be selected directly by mapping signals from the model. If no output is specified, the simulation outputs are exactly the signals of the model out ports.
Like the inputs, also the baselines can be defined by means of **Signal Editor**. There are some cases in which an assertion approach is more direct than a baseline criteria test, this will be clarified by the following table summarising all the test scenarios.

	Scenario	Baseline	Assertion
Dead state	AccPedal = exponential growth and decay over time BrakePedal = 0 SOC between 0 and 1	Null output	-
No Charge state	AccPedal = exponential growth and decay over time BrakePedal = 0 SOC <SOCmin+SOCth	BrakePedal = 0 MGUreq = 0	ICreq = min(AccPedal, ICreq_max)
Electrical Drive state	AccPedal = exponential growth and decay over time BrakePedal = 0 SOC >SOCmin+SOCth	BrakePedal = 0 ICreq = 0	MGUreq = min(AccPedal, MGUreq_max)
Combined state	MGU and IC not saturated; MGU staurated, IC not saturated; MGU not saturated, IC saturated; MGU and IC saturated	IC + MGU = AccPedal	-
Regenerative Braking state	MGU braking torque not saturated; MGU braking torque saturated	Brake+MGU = BrakePedal	-

Table 4.1: Torque Demand Controller Testing setup

As it can be noticed, some cases aim to saturate the behaviour of the MGU or IC engine. As a matter of fact, if the request coming from the driver exceeds the maximum torque which can be provided by one of the two systems, then the saturation condition takes place. To do this, the request on AccPedal or BrakePedal has been properly designed.

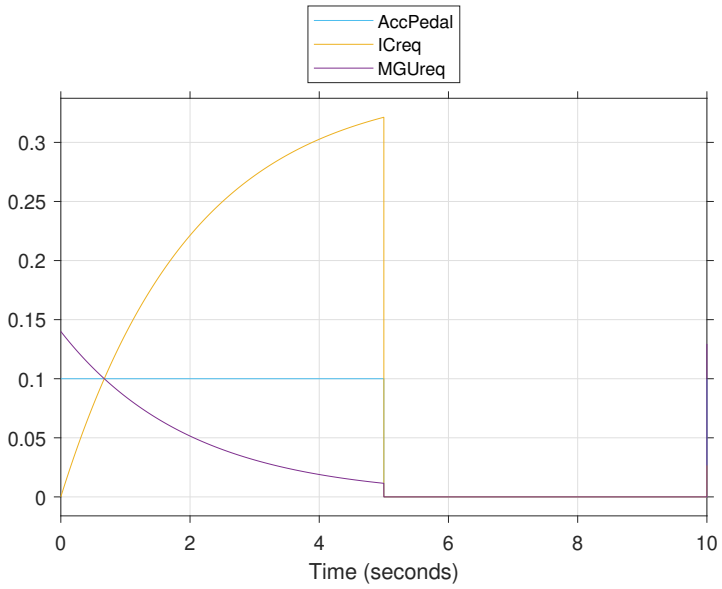


Figure 4.6: Input signal which does not cause saturation of the output

In Figure 4.6 both the IC request and MGU request do not saturate. The input coming from the driver, namely *AccPedal*, is a pulse signal of amplitude 0.1, width with respect to the period of 0.5 and period 10 seconds.

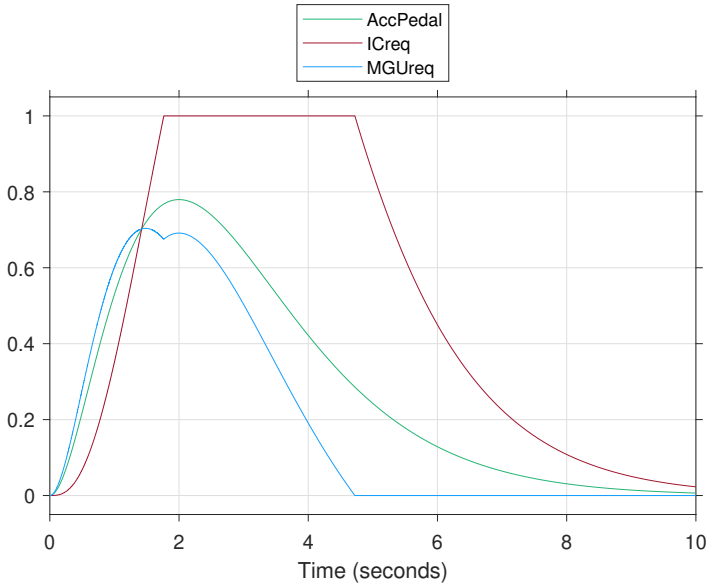


Figure 4.7: Input signal causing saturation of ICreq

In the plot 4.7, the request coming from the driver (*AccPedal*) has an exponential behaviour (rising first, falling then). It can be noticed the saturation of the IC engine. The opposite happens in plot 4.8 where the MGU saturates.

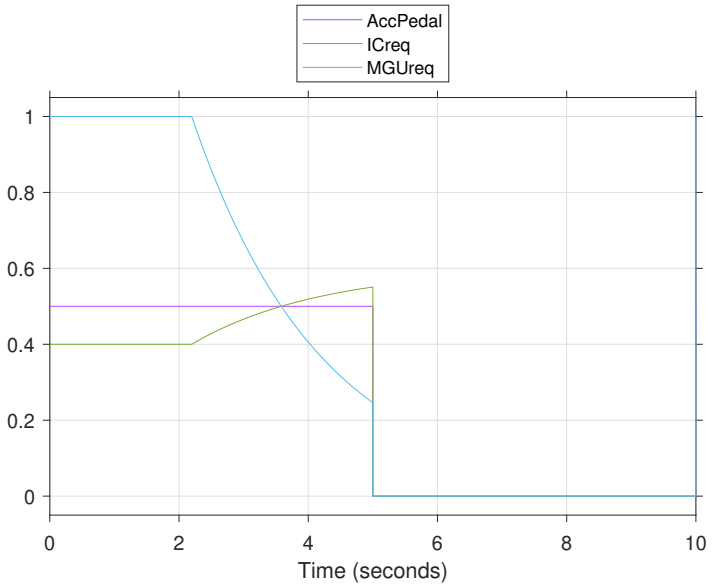


Figure 4.8: Input signal causing saturation of MGUreq

The same behaviours can be induced and tested for the braking case, when in *Regenerative Braking* state.

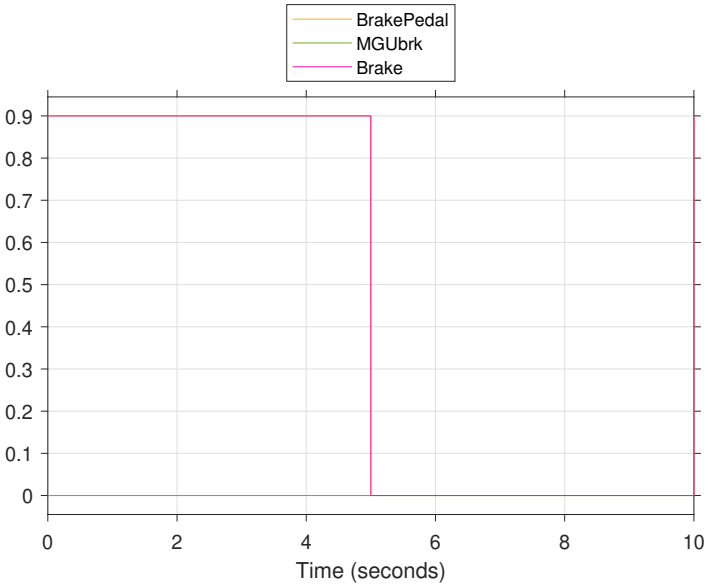


Figure 4.9: Regenerative Braking state with $SOC = 100\%$

In Figure 4.9 the battery is fully charged, so the regenerative braking is off and only usual brakes act. It is also possible to check this last condition, i.e. $MGU_{brake} = 0$, with an assertion.
For what concerns the assertions, they can be formulated in a straightforward way by means of a specific facility of Test Manager. Here all the symbols that are used need to be referred to the model signals or parameters in the workspace. The assertions are, in this case, formulated for the entire duration of the simulation time, meaning that their validity is checked at all time instants.
Once all the test cases have been defined, the **Test Specification Report** can be generated through an automated system. After that, the test can be started and, in the end, also the **Test Result Report** can be automatically generated. As previously described in the FSM subsection, there are quite a lot of options that can be included when generating reports, among which there are figure generation, author inclusion, test failure only and so on.

4.2. Integration Testing

Once the single unit has been tested, it is necessary to create an integrated subsystem, composed of FSM and Torque Demand Controller, so that the system behaviour can be verified under different working conditions.

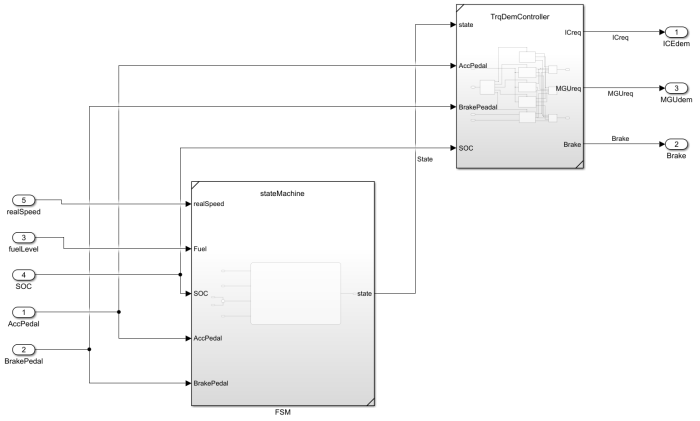


Figure 4.10: Integrated controller subsystem

As shown in Figure 4.10, the overall controller is the cascade of the two units, some inputs are common between the two units, the output of the FSM, namely the evaluated state, is given as input to the Torque Demand Controller. This last unit provides the outputs of the integrated controller.
Once the model has been created, it is possible to use it as previously described in the *Unit Testing* section, that is, by means of *Simulink Test*.
Clearly, it is impossible to cover all test scenarios, therefore only a subset of relevant cases has been considered. In particular, 8 different cases have been identified for every simulation lasting 10 seconds:

- **Electrical Drive** behaviour, when the discrimination parameter is the low speed allowing the controller to demand torque only to the MGU.
- **No Charge** behaviour, when there is Acc_{Pedal} request greater than zero and the battery is low ($SOC < SOC_{min}$).
- **Electrical Drive** behaviour, always at low speed, with the saturation of the MGU request in terms of torque because of the higher Acc_{Pedal} request after 5s.
- **No Charge** behaviour, with the saturation of the ICE request in terms of torque because of the higher Acc_{Pedal} (higher with respect to the maximum ICE acceleration) request after 5s.
- **Combined** behaviour and transition to **Regenerative Braking** state when, after 5s, $Brake_{Pedal}$ gets higher than zero. In this case the car is traveling at high speed, so that the controller cannot work in Electrical Drive. Both fuel and SOC are above the minimum for working.
- **Dead** state behaviour, when SOC and fuel are below the minimum and there is a request from Acc_{Pedal}. The system must provide zero output in this particular condition but must also be able to switch to **Regenerative Braking** when there comes $Brake_{Pedal}$ request.
- **Increasing speed scenario**, a typical use case: the vehicle is driving in a town environment at low speed, then gets in an extra urban scenario, travelling at higher speed. Here the controller shall be able to switch from Electrical drive to Combined mode, demanding torque to both motor and engine.
- **Recharging scenario**: the vehicle starts with an almost empty battery, using only the IC engine. During the drive, the battery recharges through **Regenerative Braking** and when the State of Charge is sufficient, the controller can require torque to the MGU.

All the test cases have been created by means of **Signal Editor** tool and fed as inputs to the subsystem. The type of test is now set to *Simulation Test* and the checks have been performed through assertions. The assessment can either be logical or triggered by signals. In this case the event triggered checks are suitable for the purpose of these tests. Indeed, once the trigger, like the input Acc_{Pedal}, has been detected the check for signal response is done.

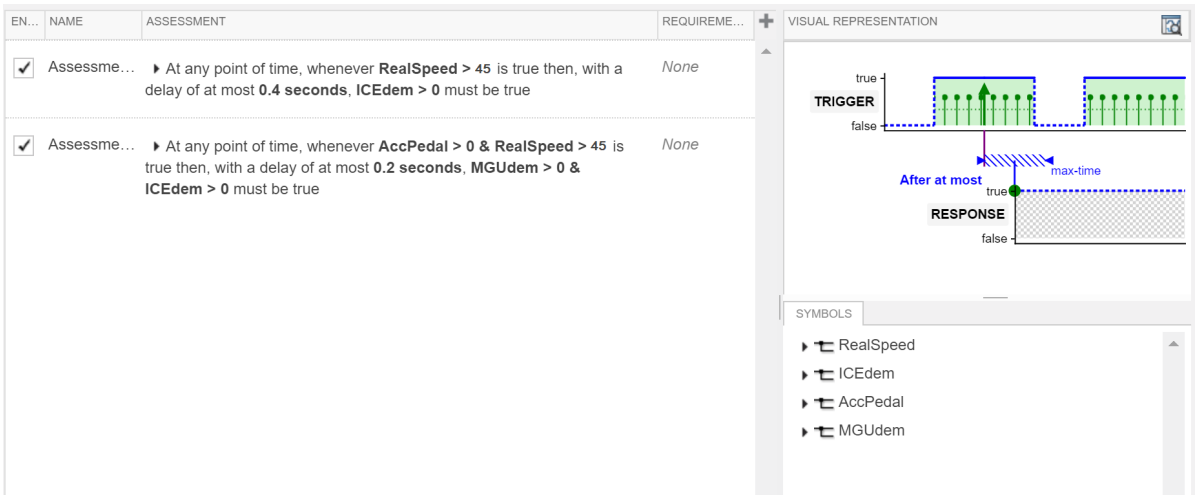


Figure 4.11: Integrated controller: assertion example

In the example shown in Figure 4.11 the two assertions are triggered every time the signals in bold, namely **RealSpeed** and **AccPedal**, satisfy given conditions. Then, a delay has been considered because of the dynamic nature of the integrated subsystem. From previous discussion indeed it has been highlighted the fact that the controller implements filters and the FSM is a discrete unit sampling the inputs with a given periodicity.
Moreover, when performing assertions, it is necessary to map symbols to signals inside the considered model. If there is any problem with mapping, the symbol causing the issue becomes red in the symbol window.
In the end, by the moment all test cases have been defined, the Test Specification Report can be generated, again automatically. Once more, after the test have been executed, also the Test Result Report can be automatically created.

4.3. Model-in-the-Loop Simulation (MIL)

MIL provides the very first closed loop simulation, in which both the plant and the controller are simulated by their respective models. This procedure allows simulating how the controller behaves once integrated into the plant, even before the code generation, thus ensuring faster feedback of the design phase.

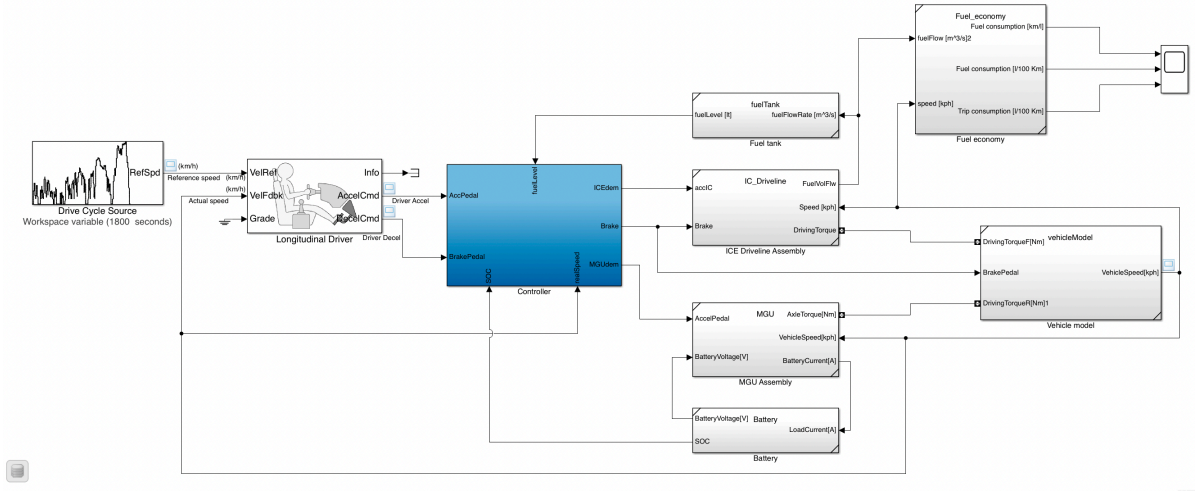


Figure 4.12: Simulink model for Model-in-the-Loop Simulation

Figure 4.12 shows the Simulink model in which the controller has been integrated into the plant, such that it is responsible for managing the torque allocation between the IC engine, MGU and brakes, depending on the driver request and on the current state of the vehicle.

MIL simulation has been used to verify the overall behaviour of the controller, including:

- The capability of the vehicle to follow the drive cycle, meaning that the torque management strategy does not ruin the driving performance (Figure 4.13).
- The torque request, depending on the state of the vehicle (Figure 4.14).
- The assessment that the weighted sum of the IC engine, MGU and braking torque requests is equivalent to the actual driver request (Figure 4.15).

The following figures show some MIL simulation results focusing on a sub-interval of the drive cycle ([1000; 1470] seconds) for better visualization. For the complete plots, please just run the MIL simulation.

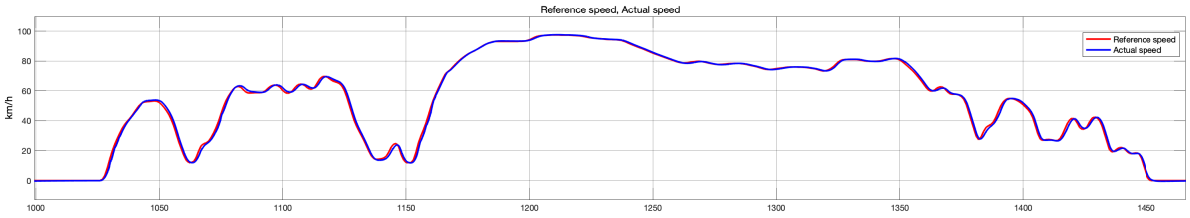


Figure 4.13: Comparison between drive cycle reference speed and actual vehicle speed

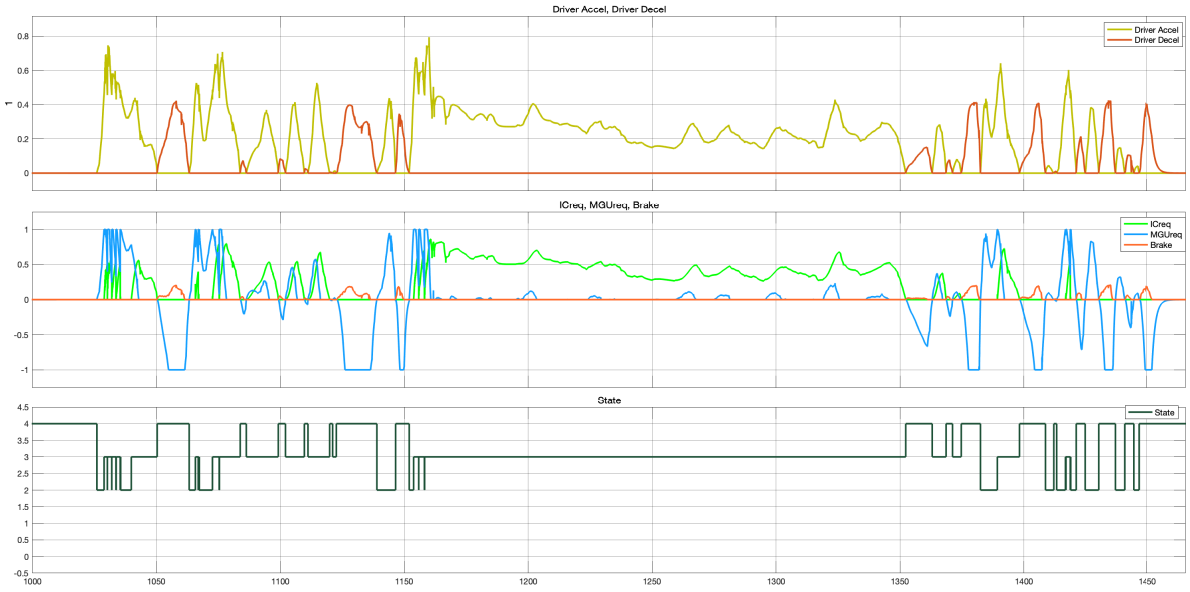


Figure 4.14: Driver acceleration and braking command (top); controller outputs IC engine, MGU and braking request (middle); vehicle state (bottom)

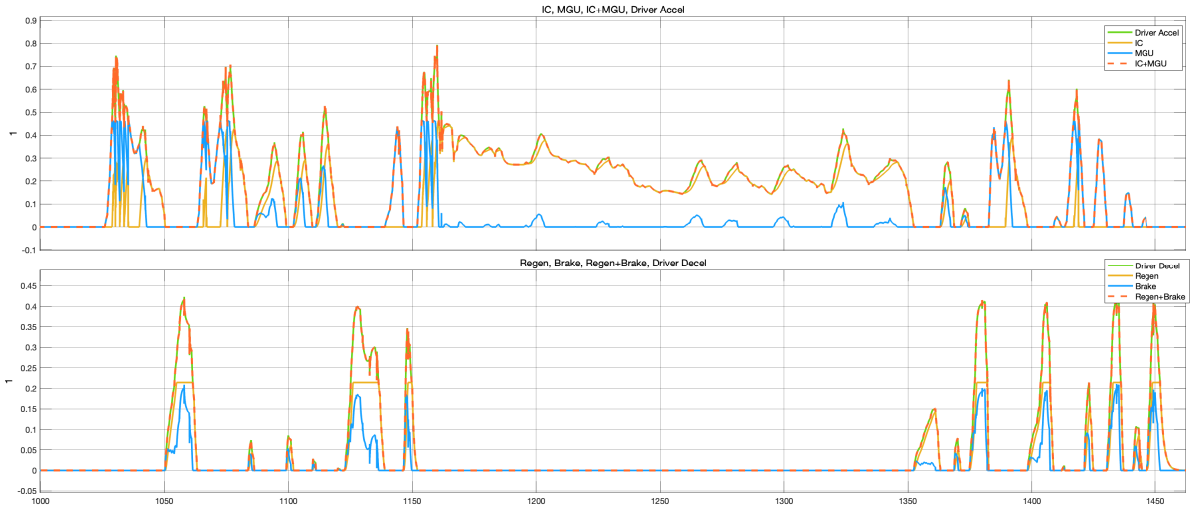


Figure 4.15: Direct comparison between driver command and controller outputs: the weighted sums of the IC engine, MGU and brake request are equivalent to the driver acceleration and braking commands

Moreover, Figure 4.16 shows the behaviour of the battery state of charge (SOC) during the WLTP cycle. It is noticeable that even though the electric motor is mainly used at low speed, the control

[0]For a better visualization it is reported just a sub-interval of the whole drive cycle.

strategy allows for the MGU to supply torque to help the ICE even at speeds higher than the maximum allowed for a full-electric drive.

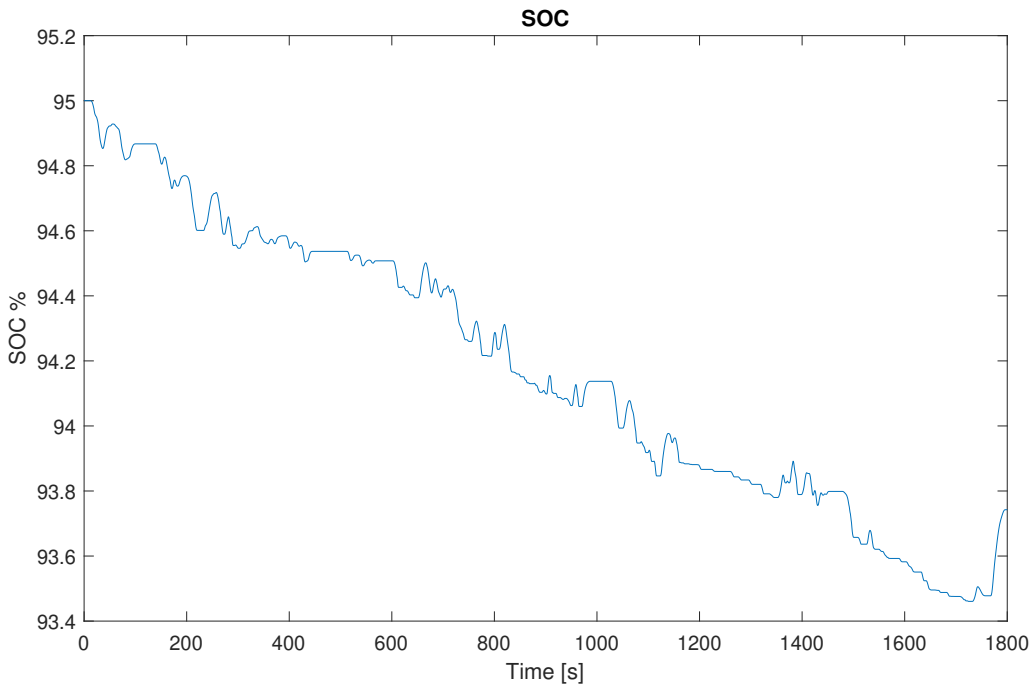


Figure 4.16: Battery SOC over the drive cycle

4.4. Automatic code generation

Embedded code generation is a fundamental step that is changing the way engineers work. Instead of writing thousands of lines of code by hand, engineers are automatically generating their production code to increase productivity, improve quality and avoid errors while writing the code.
A first consideration is that, while the design of a system consists of a float model, on the other hand the implementation into an embedded system uses fixed-point arithmetic. Automatic code generation helps in creating source code, taking into account the target architecture, also optimizing the code for that. In addition, this code can easily pass the compliance test with coding standards, like MISRA C.
To generate C code from a Simulink model or subsystem it has been used the **Embedded Coder** tool which generates readable, compact, and fast C and C++ code for embedded processors used in mass production. It is an extension of Simulink Coder with advanced optimizations for precise control of the generated functions, files, and data. These optimizations improve code efficiency and facilitate integration with legacy code, data types, and calibration parameters. An important aspect is that Embedded Coder offers built-in support for AUTOSAR and MISRA C which suits this application.

4.4.1. MISRA C Compliance

To check that the controller model has a likelihood of generating a C code that is MISRA C compliant, the **Code Generation Advisor** has been used. Some checks are automatically run before code generation. After that, the system notifies which checks have been passed also with the possibility of generating an HTML report.

4.4.2. Code generation using Embedded Coder

The passages followed in order to generate the code are described here:

1. Open the model containing the block for which the code needs to be generated
2. Open the Model Configuration Parameters dialog box

3. Select the Code Generation tab
4. To select a system target file, in the Target Selection pane, click Browse. It can be generated code for a specific target environment or general-purpose architecture
5. Select Embedded Coder, also indicated as Embedded Real-Time (ERT) system target file and click Apply
6. In the Code Generation Advisor pane, click Set Objectives. This is a very important stage, as the coding standard compliance can be checked right here. There are also other objectives that can be selected. Moreover, you can select and prioritize a combination of objectives before generating the code
7. In the model window, initiate code generation and the build process for the model by right clicking on the component and going to C/C+ Code Generation, then clicking on *Build*
8. View the code generation report. The report includes the .c file, associated utility, header files (.h), traceability and validation reports.

4.5. Software-in-the-loop Simulation (SIL)

With SIL simulation, it can be verified the behaviour of the controller source code on a target host computer. Thanks to the **Embedded Coder toolbox** there are different ways to perform a SIL simulation. The first one consists of creating a test harness in SIL mode using it to perform several actions:

- View and compare logged data and signals using the Test Manager and Simulation Data Inspector, which is useful to check how the controller works when running on simulation or on a host architecture.
- Use built-in tools for these test-design-test workflows, checking SIL equivalence or even updating the SIL to the latest model design
- Verify the generated source code, related to the controller.

The second method is represented by the creation of a SIL block. This is the strategy that has been followed in this project and it is described in detail from now on.
Firstly, it is necessary to have the Embedded Coder toolbox installed together with Simulink Coder. In the model settings of the Simulink Model it needs to be specified the use of this Embedded Coder in the **Code Generation** section; once having set this parameter it is possible to check on **hardware settings** the host architecture on which the automatically generated code for the controller will be running.

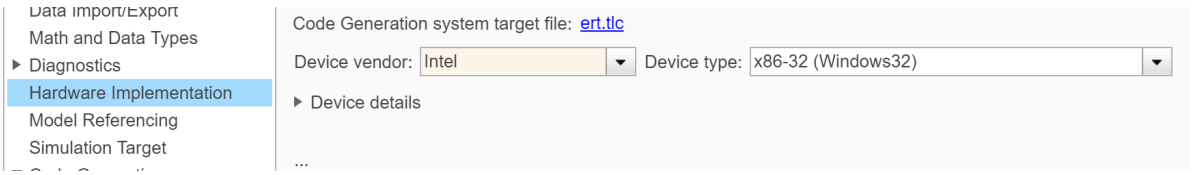


Figure 4.17: Target Architecture selection

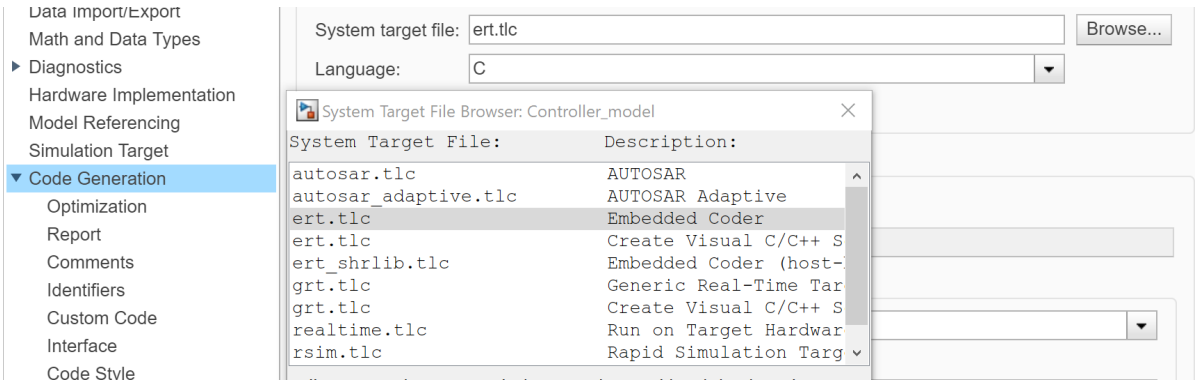


Figure 4.18: Embedded Coder selection

As shown in Figure 4.18, the system target file shall be *ert.tlc*. This procedure is the same also for the Processor-in-the-Loop block generation.
Secondly, a fundamental configuration, which is kind of hidden in Simulink, is the SIL block generation. It can be issued by checking a specific flag in the **Code Generation** section (verification subsection as shown in Figure 4.19), always in model settings. After these operations, the model settings window can be closed to keep the focus on the system under test.

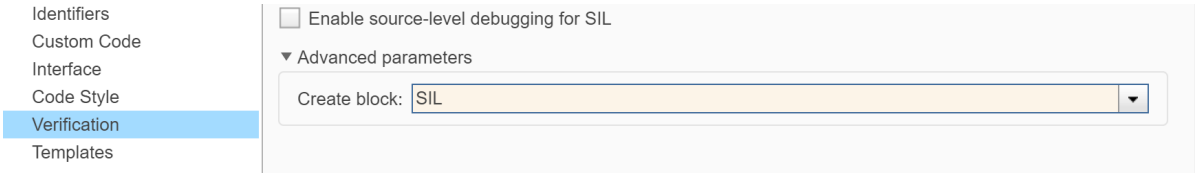


Figure 4.19: SIL block generation flag

In the Simulink model, once identified the system or subsystem that needs to be tested in SIL mode, it should be a good practice treating it as an atomic unit, by checking the correspondent flag in **Block parameters**. This enables the Function Packaging parameter on the Code Generation tab, useful for the reuse of generated code.
After that, by right clicking on the unit, in the section of code generation, the build functionality can be issued. In the Controller dialog box, by clicking *Build* the creation of SIL version of the Controller subsystem block will happen in a new model.

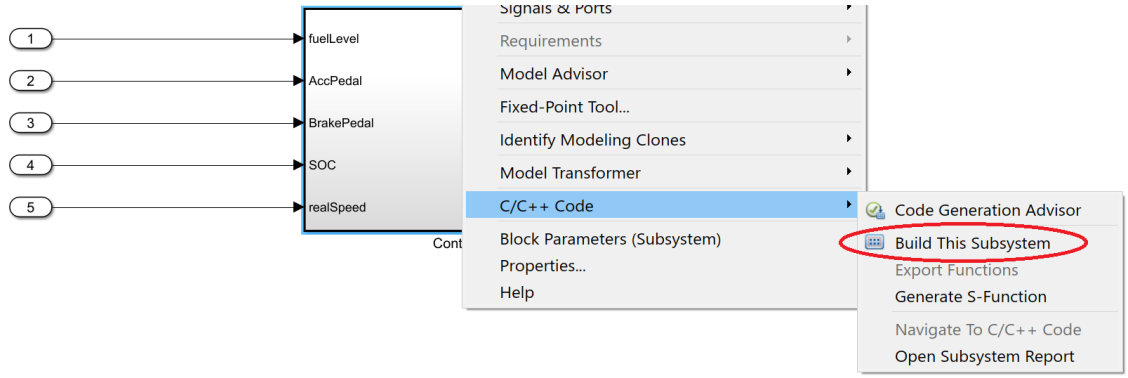


Figure 4.20: Subsystem C code building

This new model, like the one shown in Figure 4.21 can then be exported inside the original Simulink model, also containing the simulated plant which will still be running as a Simulink simulation.

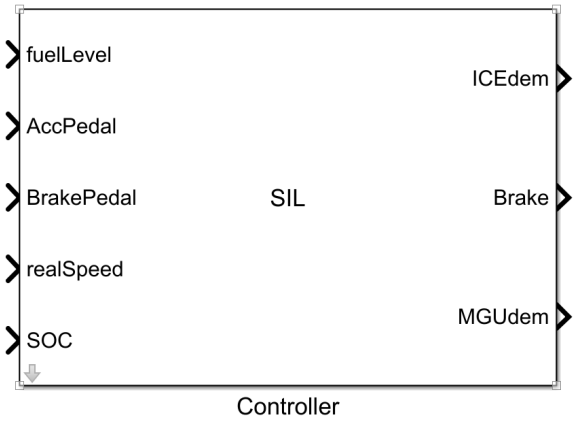


Figure 4.21: Controller SIL generated block

After the SIL block has been generated, a comparison simulation has been carried out. The entire plant has been duplicated and the error in speed between the SIL block and the controller designed in Simulink has been recorded (Figure 4.22)

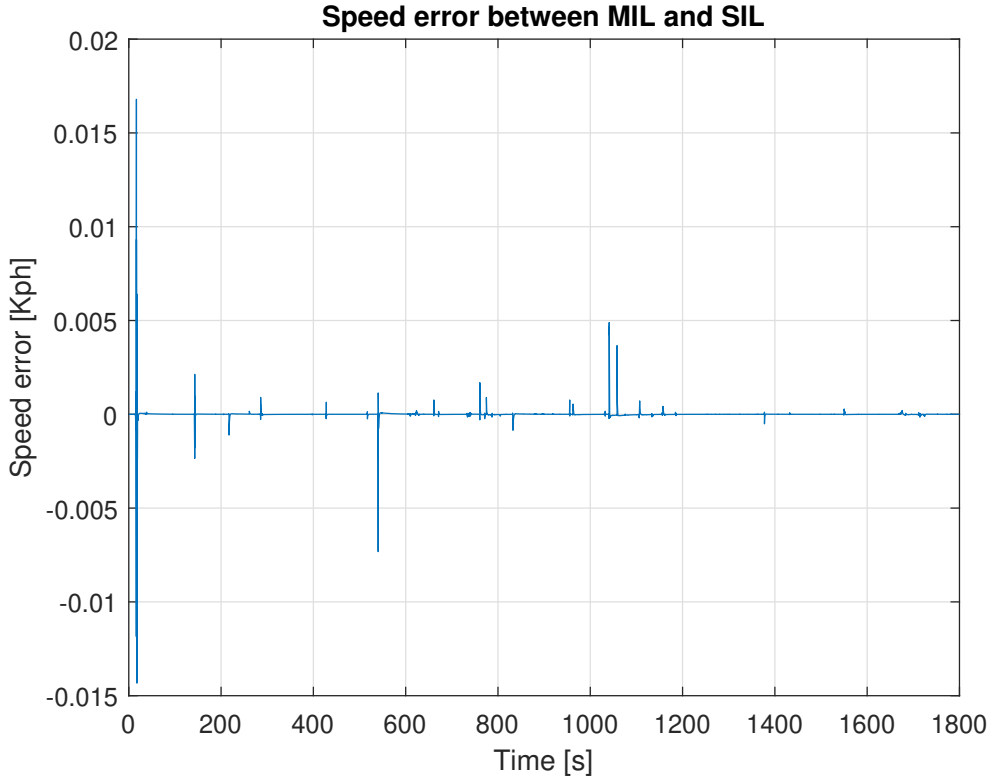


Figure 4.22: Error speed

4.6. Processor-in-the-loop Simulation (PIL)

Processor-in-the-loop Simulation (PIL) provide a further testing phase in which the controller code runs directly on the actual target, whereas the vehicle model is still simulated. The main benefit of PIL is that the generated code is target specific, thus allowing to test how the architecture, optimization and coding style could affect the final behaviour.

The chosen MCU architecture is the 8-bit ATMEGA328p microcontroller and the Arduino Uno evaluation board has been used for interfacing the MCU with Simulink. Similarly to SIL simulation, the Embedded Coder toolbox gives the possibility to generate the code and a PIL block, that runs on the specified target instead of the host computer. Due to the limited numerical precision of the ATMEGA328p, it cannot handle 8-Bytes double data type, thus the Simulink model is first adapted to deal only with single data types. Figure 4.23 shows the connection of the generated PIL block: it is worth noting that the single data type controller model is exploited for the code generation, whereas the comparison is done between the PIL block output and the double data type model.

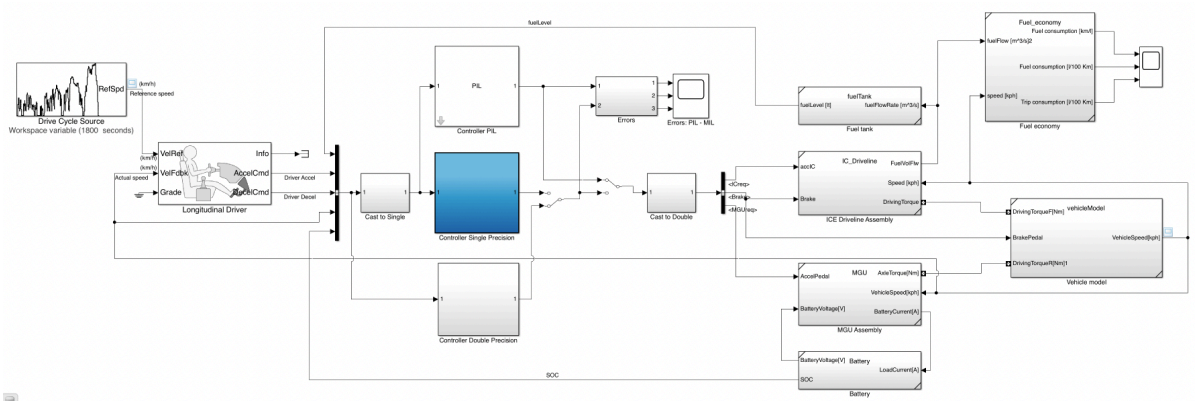


Figure 4.23: PIL simulation setup

It is possible to evaluate the numerical equivalence between the model output and the PIL block output by running the simulation and computing the errors: the order of magnitude is 10^{-7} for all three outputs (Figure 4.24).

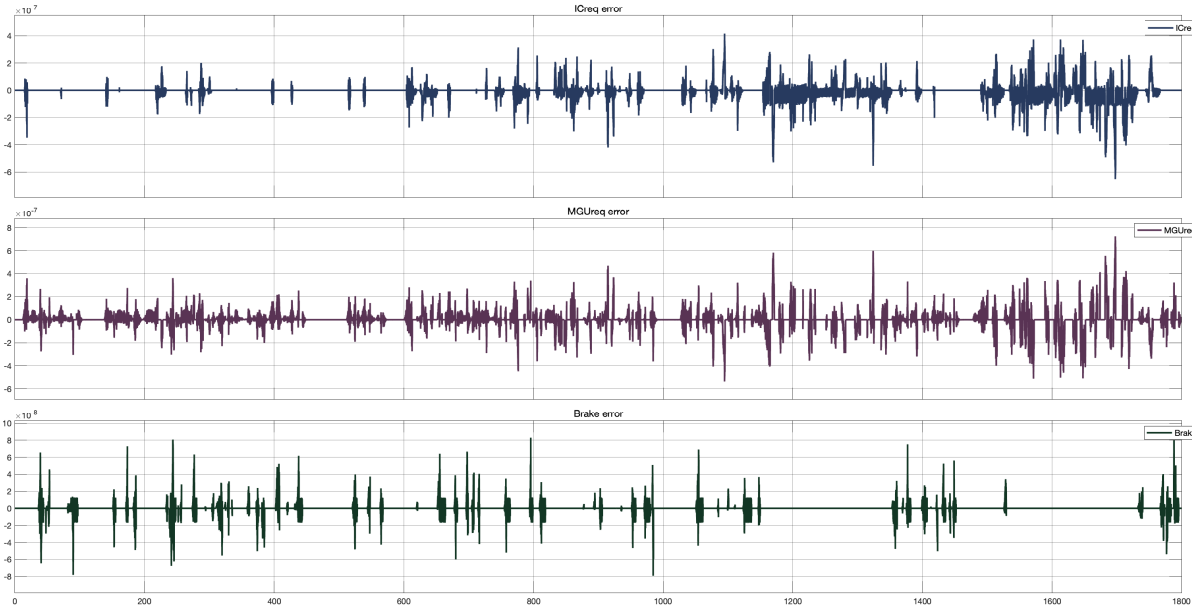


Figure 4.24: Error between the double numerical precision Model output and PIL output

Figure 4.25 shows the vehicle speed behaviour during the drive cycle when the target processor is

inserted into the loop.

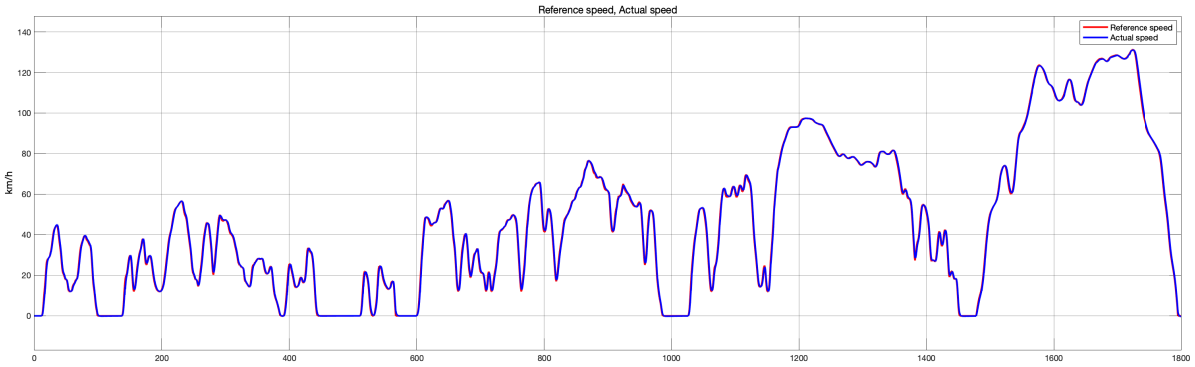


Figure 4.25: Vehicle speed in the PIL simulation

5

Simulation results and possible developments

Nowadays the design phase of any controller or ECU has shifted to the Model-based design. This incredibly powerful approach, together with software such as Simulink, enables new ways to develop state of the art devices. Throughout the project different aspects of the V-model have been analysed. The ability to simulate extremely complex systems and interconnections is fundamental to satisfy each step of the model. The Simulink blocks that have been used for the modeling are best suitable for this approach given the fact that they were designed to be properly configured with empirical data gathered from real life scenarios. The choice of this project is not a random one. In fact, PHEVs are the most suited vehicles for the near future because they can harness both internal combustion engines and motor generator units, discarding the disadvantages of both the technologies. To properly compare the behaviour of the controller, two vehicles have been simulated:

- **Vehicle A**: it is equipped with only the internal combustion engine, the weight is in compliance with the average weight for a car in the market, while the driver is kept the same (PI controller)
- **Vehicle B**: this is the plug-in electric vehicle on which the controller is based. Considering the fact that the car features all the components necessary to perform the electric drive, the weight is approximately 36% more with respect to the vehicle's A. Needless to say that the car also features the hybrid controller designed for the project

As a matter of fact, the controller - thanks also to its strategy - can successfully limit the drawbacks of the endothermic engine thus reducing the fuel consumption over the WLTP cycle trip. From Figure 5.1 it can be possible to observe that, even with the modified weight, vehicle B consumes less fuel than vehicle A (the average trip consumption is given in $L/100Km$). Moreover, it is noticeable that the biggest difference in the consumption can be found at low speeds (i.e. the MGU is working in vehicle B), whilst the consumption at higher speeds is higher in vehicle B because of the weight difference.

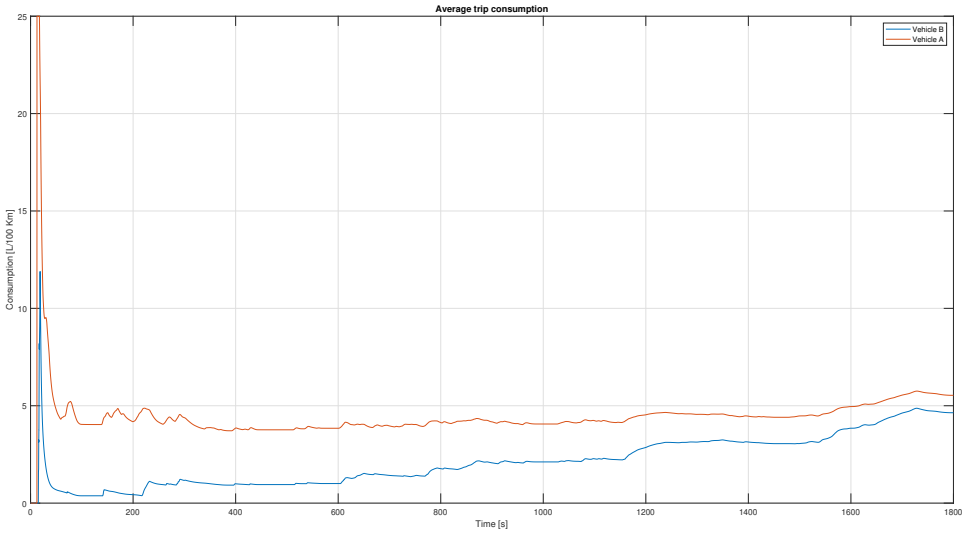
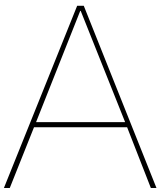


Figure 5.1: Average trip consumption

The controller that has been designed for this project could be easily modified to integrate it into a multi-ECU system. By simply adding CAN peripherals into it, it could be possible to also simulate the communication efficiency between all the nodes of the vehicle. Moreover, keeping in mind the future trends in automotive and legislation, the implementation of a V2X emulated ECU could bring new variables into play when considering the control strategy. In fact, more and more municipalities are moving toward a strategy focused on limiting the use of fossil fuel in the city centre and its surroundings, thus making the PHEV the best choice.



Model numerical parameters

Vehicle Parameters	Value	Unit
Vehicle mass	1500	Kg
Wheel radius	0,35	m
Diff ratio	3,4	%
Battery capacity	11	KWh
Driver Parameters		
P	15	
I	0.5	
Engines Parameters		
MGU Max Power	50	kW
SOC max	99	%
torque MGU max	150	Nm
torque IC max	175	Nm
Tau IC	2	
Tau Regen	1	
FSM & decisional parameters		
SpeedEDmax	45	kmph
fuelMin	0,33	l
fuelHys	0,17	l
SOCth	5	%
SOCmin	10	%
maxAccICE	TrqICMax/TrqMax	
maxAccMGU	TrqMGUMax/TrqMax	