POLITECNICO DI TORINO

Master's degree in Automotive Engineering

Real-Time Implementation of an Electric Vehicle Thermal Management System

Candidate:
Mattia SCALISI

Supervisor(s):
Prof. Ezio SPESSA
Prof. Daniela Anna MISUL
Dr. Federico MIRETTI

MARCH 2025

Abstract

The importance of the Battery Thermal Management System (BTMS) is paramount within the context of the life cycle and autonomy of hybrid and electric vehicles, considering also the increasing importance of the EV market. In this context, this work explores the feasibility of a real-time application of a BTMS, developed in the Simscape environment by Mathworks. In a previous work, the PID Controller originally used by Mathworks was replaced with a much more efficient Model Predictive Control (MPC). The MPC manages the compressor of the refrigerant cycle and is fundamental in battery temperature management. Within this work, the model was first adapted to a previous Matlab version (2020b), and various components were completely modified, such as the compressor and the fan. The model was then tested with a proper solver in a fixed time-step before carrying out different real-time simulations. These were conducted using the dSPACE environment with different Hardware setups. Simulations were first performed with the whole system uploaded to a single Hardware unit and then, by ensuring CAN communication between two different dSPACE Hardware units, one was used solely for the Controller and the other for the Plant. The results were then analysed in this work, comparing them to the offline ones, and considerations on CAN payload and control system computational burden were made. The control system turned out to be feasible, and new developments on the full model (with HVAC) could be carried out in the future.

Contents

List of Figures

List of Tables

1 Introduction

In the context of the actual automotive industry, electric cars continue to make progress towards becoming a mass-market product in a larger number of countries. Despite the widespread concerns about battery technology, inflation, and the rising prices of metals, their sales remain very strong. [1]
Moreover, with the increasingly stringent policies on greenhouse gas emissions, every other car sold in Europe by 2035 is set to have zero emissions, so electrification is at the center of research in the automotive field as a technology that can help achieve these goals.
Batteries, the energy storage system used in electric vehicles, are a relatively new technology; this means that there are various challenges to be addressed and significant room for improvement. The main issues are related to their capacity and recyclability, but one of the biggest drawbacks linked to Lithium-Ion Batteries (LIB), currently the most widely used technology in the market, is the risk of overheating and safety hazards, including thermal runaway with a series of chain reactions that can lead to a fire.
Moreover, LIBs perform much better within a temperature range of 15°C to 35°C, with a temperature difference between the cells that does not exceed 5°C. Failure to comply with these rules can lead to premature degradation of the battery and reduced performance. [2]
Battery Thermal Management Systems (BTMS) ensure the safe and efficient use of batteries, keeping them in an optimal range for extended battery life and reliable operations; they play a vital role to protect batteries from the negative impact of heat generation and increased temperatures.[3] At the same time, the use of BTMS can reduce the life of the battery within a single mission, so it is essential to manage the proper components with it, like the compressor in this work, using them only when strictly necessary.
It has thus become increasingly urgent to improve current control systems for these technologies to reduce their energy consumption further. In this context, to manage the compressor power within a BTMS, a Model Predictive Controller (MPC) has been developed in place of a

reactive Controller: this configuration has shown better performance and improved battery efficiency.

1.1 Thesis Outline

This Master's Thesis aims to continue working on the **MPC-based Thermal Management System** by testing it in a real-time environment through the dSPACE systems and Software.
The core objective is to evaluate how effective the Controller can be in pursuing the target temperature even with all the constraints linked to an operational environment.
Different configurations have been tested to progressively study the critical aspects of different real-time configurations, ranging from a full model simulation on a single Hardware to a much more complex setup with a second Hardware and a CAN communication.
Each simulation of this work has distinct goals and foundations. After the system is ready for a real-time simulation, the study starts with the single model simulation, which has as its aim to show the feasibility of real-time application of the full BTMS model, with its bottlenecks and high computational load, to pave the way for the other developments. Then, the Controller and the Plant are divided between two different tasks to assess how the division of the computational load into more than one memory stack can improve the results. Moreover, different solvers are used and compared for the Controller.
Afterward, the core of the work lies in the HiL simulation: it is essential to analyze the performance of the Controller, the element that will work in a real-time context, both in terms of errors and latencies introduced by the CAN bus communication.
In the last simulations, the same HiL setting is used with a much more robust model, that uses custom components (compressor and fan) to improve the representation of the system behavior.

2 Thermal Management System

In this chapter, the Simscape/Simulink model used in this work will be presented. The work is based on a **Thermal Management System of a BEV (Battery Electric Vehicle)** that has been downloaded directly from Mathworks documentation.[4]
This model originally used a Reactive logic of control that was then modified into an AMPC, Adaptive Model Predictive Control.

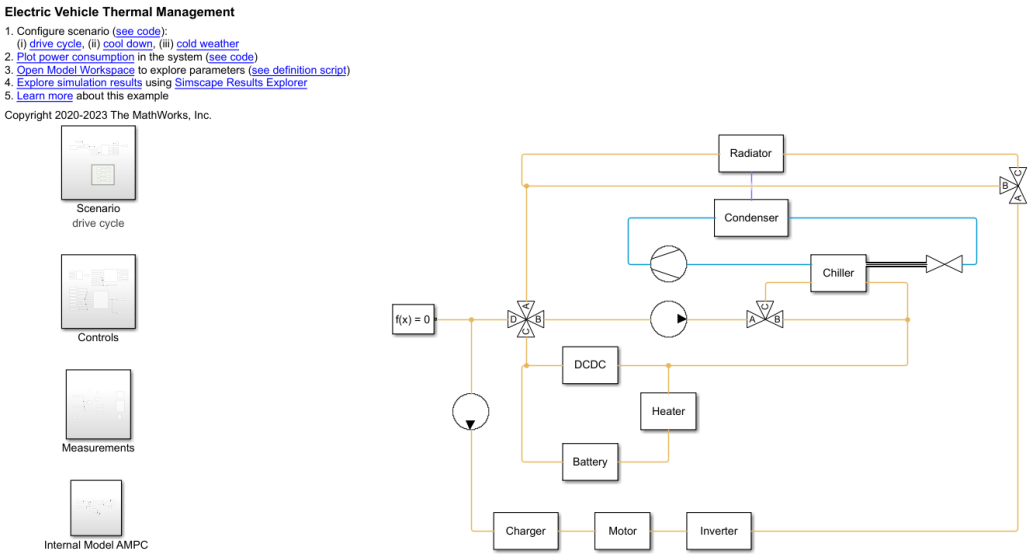


Figure 1: Full Model

2.1 Plant

The system originally consisted of two liquid coolant loops, a refrigerant loop, and a cabin HVAC loop. The thermal loads are the battery, the powertrain, and the cabin. However, in this case, the HVAC loop and the cabin have been left out as a continuation of the previous Master Thesis work since the MPC was implemented for this configuration. To understand the modalities in which the system works, the valves are the components to start with. There are two **three-way valves** and one **four-way valve**. As far as the four-way valve is concerned,

it makes the system work in series or parallel. In parallel mode, the system works as if there were two different refrigerant circuits (A-D, B-C connections), while in serial mode, it works as a single circuit (A-B, C-D).
As far as the three-way valves are concerned, the upper one is the radiator bypass and the other the chiller bypass.

2.1.1 Scenarios

In cold weather, the system works in serial mode, the chiller and the radiator are bypassed while the motor heat is used to warm the batteries. There is also a heater in series with the battery. Cold weather was anyway not considered in the first work because the main focus was the cooling of the battery for temperatures higher than 20 °C.
In warm weather, when the temperature rises above 25 °C, the radiator bypass is switched off so that both the batteries and the motor are cooled by the radiator in serial mode.
When the weather is hot, the coolant loop switches to parallel and separates the system; while one loop cools the powertrain using the radiator, the other cools the battery through the chiller in which a refrigerant liquid flows due to the compressor. The main focus of the MPC in this model is indeed the control of the compressor that is by far the most power-draining element within the whole thermal management system.

2.1.2 Components description

To better understand the system behavior it is necessary to describe its most significant parts. There are three different liquids: the one flowing in the yellow domain is glycol water, the one flowing in the blue domain is a refrigerant fluid (R134a), and the one flowing in the purple domain is moist air.

These are the components in the **Thermal Liquid** and **Moist Air** domains (yellow and purple lines):

- **Pumps:** Two pumps drive the coolant loop, one is used for the charger/motor/inverter and the other one for the battery/DC-DC converter. The pumps are controlled by a velocity input and respond to a reactive logic. They are modeled with pump blocks that make the mechanical and thermal domains interact. The electrical energy necessary to make them work is considered through mechanical efficiency.

- **Charger, Motor, and Inverter:** These components are not modeled functionally but from a thermal point of view. There is a coolant jacket placed around the component, represented by a heat flow rate source, which models the heat generated by Joule effect and a thermal mass. The goal is to depict the conductive and convective heat exchange between the thermal liquid and the components through the coolant jacket. The thermal liquid absorbs heat from the components.

- **Battery:** The battery is modeled by considering four battery packs connected in series and a coolant jacket that models the conductive/convective heat exchange for each of them. The thermal liquid absorbs heat from the battery. Then, there is the electrical circuit where a certain voltage is imposed on all the packs, and a certain current is required by the battery based on the performance requirements. Within each pack, the lithium cell is modeled by a 1RC equivalent circuit model.

- **Radiator:** The radiator is a component that primarily rejects heat to the environment. It is used to cool the thermal liquid involved in the loop in warm weather whenever the temperature exceeds a certain threshold (25°C), and the valve of the radiator bypass is opened. The radiator is used to cool all the elements of the circuit, such as the battery and the motor. The radiator is modeled as a cross-flow heat exchanger.

- **Fan:** The fan is the element that causes the fluid (moist air) to move within the proper channels when the ram pressure of the ex-

ternal air (pressure given by the reciprocal speed) isn't high enough to enhance the heat transfer. The description of the fan used in the first model will be explained in detail later.

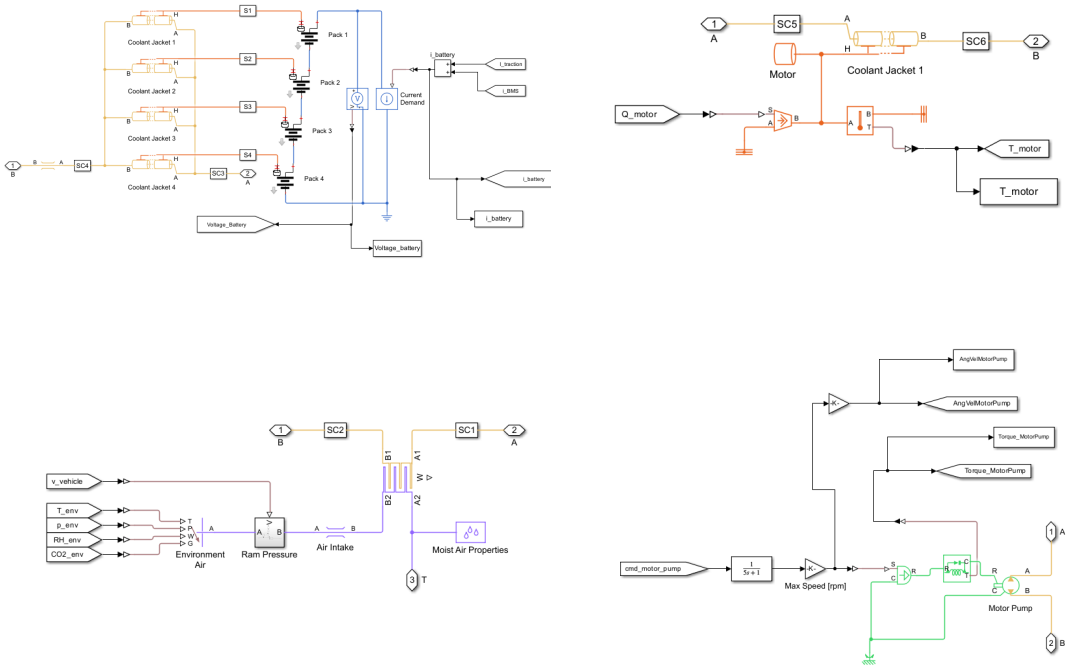


Figure 2: In order: Battery, Motor, Radiator, Motor Pump

Then the components in the **Refrigerant** domain (blue line) can be highlighted. The blue line identifies a refrigeration cycle [5] that contains the compressor, which, being the energetically most demanding component, is, as mentioned, controlled by the MPC (differently from the other elements controlled by a reactive logic). In this circuit, the refrigerant is heated to cool the battery through the thermal liquid.

- **Condenser:** The condenser is a heat exchanger that allows the heat transfer between the refrigerant and the moist air coming from the outside through the radiator. Here the refrigerant in superheated vapor form is cooled down at constant pressure until

reaching the saturation temperature, then the condensation happens and the fluid is further sub-cooled.
The main component in the subsystem is the heat exchanger but there is also the fan in the same subsystem. A proper Simscape cross-flow heat exchanger block is used for its modeling.

- **Expansion Valve:** The expansion valve expands the fluid from condensing to evaporating pressure through an isenthalpic transformation. Through the expansion valve, indeed, a pressure drop occurs.
  It is modeled by using the proper expansion valve block present in Simscape.

- **Chiller:** The chiller is the evaporator of the cycle and makes possible the heat exchange between the refrigerant and the thermal liquid. The liquid from the expansion valve is heated at constant pressure up to evaporation temperature. The refrigerant is completely vaporized and then superheated to avoid mechanical damage to the compressor due to the liquid droplets.
  Here a cross-flow heat exchanger to model the chiller is used as seen before.

- **Compressor:** The compressor creates the pressure increase from the evaporator to the condenser and can be used to modify the cooling capacity by adjusting the refrigerant mass flow rate through the evaporator. This happens by controlling the input velocity and, by it, the correspondent mass flow rate.
  The description of the compressor used in the first model will be explained in detail.

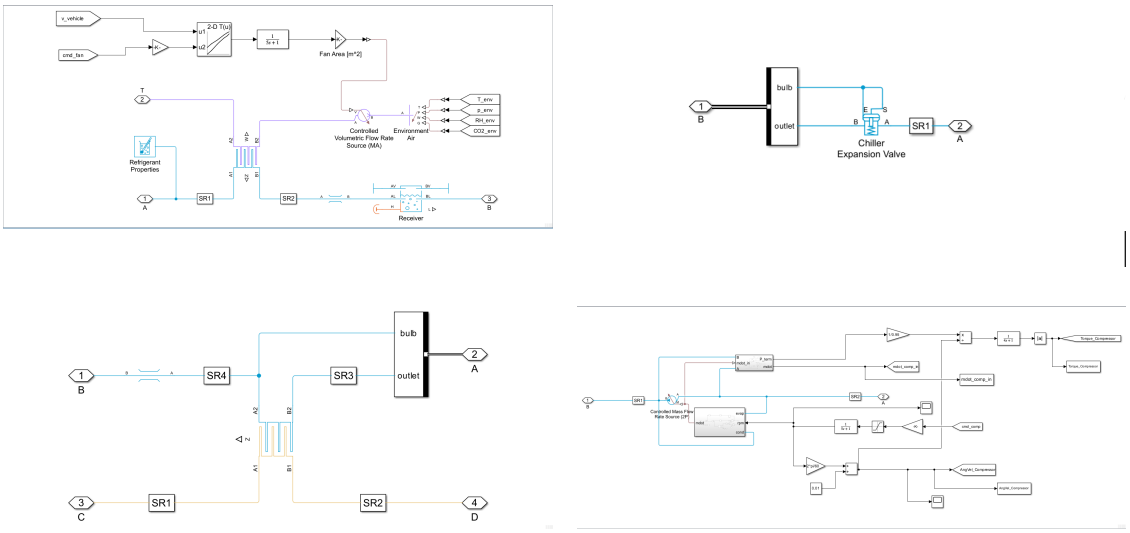


Figure 3: In order: Condenser and Fan, Expansion Valve, Chiller, Compressor

2.2 Downgrade to Matlab 2020b

To set up the real-time simulation and to make the model compatible with the licenses available for the dSPACE environment, a downgrade to Matlab 2020b was mandatory.

2.2.1 Compressor and Fan

With the downgrade, some components had to be redesigned according to the old Matlab version with fewer functions.
The two elements that were changed the most are the compressor and the fan, because the related blocks weren't available on Simscape 2020b.
At first, the compressor was modeled based on the compressor system that was used in the original model provided by Mathworks, by consulting the documentation before cited, but relative to Matlab version 2020b. The component is based on Simscape block **"Controlled Mass Flow Rate Source"** where the mass flow Rate is given by a

2D LookUp Table. It is obtained as a function of the compression ratio and of the input speed provided by the control system. Then the related torque is simply found by the equation :

$$T = \frac{\dot{m}(h_{\text{out}} - h_{\text{in}})}{\eta_{\text{mech}}\omega} = \frac{P_{\text{mech}}}{\omega} \tag{1}$$

This provides the actual compressor torque to the Controller: considering thermal power and mechanical efficiency, the mechanical power is obtained, and, dividing by the speed, the torque is given.

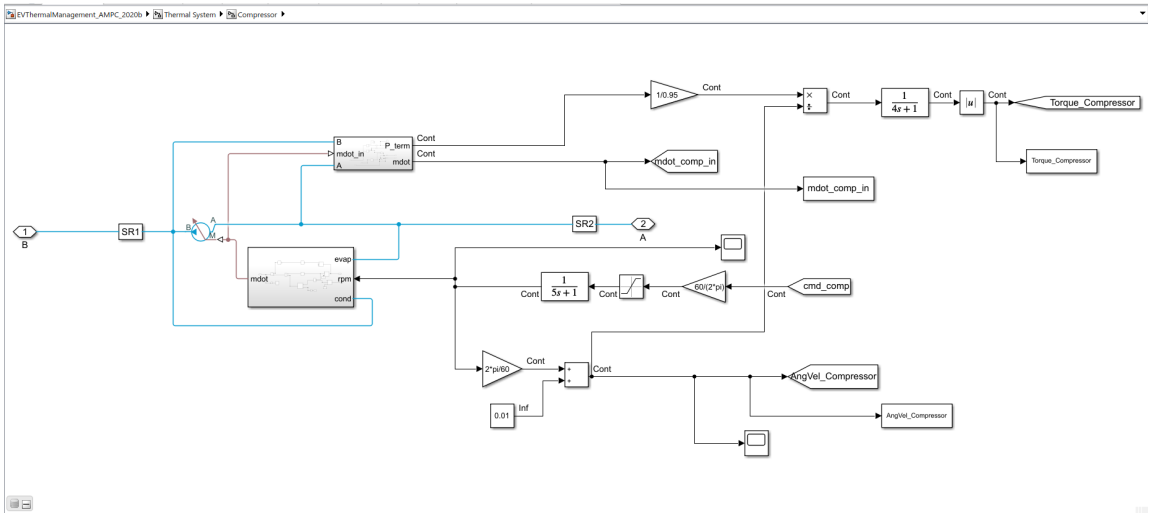


Figure 4: Compressor Model Matlab 2020b

The compressor behavior seems reasonable and works according to its purpose, lowering the battery temperature when is heading towards 30 °C.
Then, also the fan model was created starting from the one given by Mathworks documentation. It uses a **"Controlled Volumetric Flow Rate Source"** whose volumetric flow is chosen through a 2D Look-up Table as a function of the vehicle speed and the command of the fan, which is controlled in terms of velocity. While the parameters remain unchanged in this first phase of the study, the fan doesn't work right now (and this is a problem that will be addressed later since the fan

isn't the object of the control system and won't affect the real-time implementation).

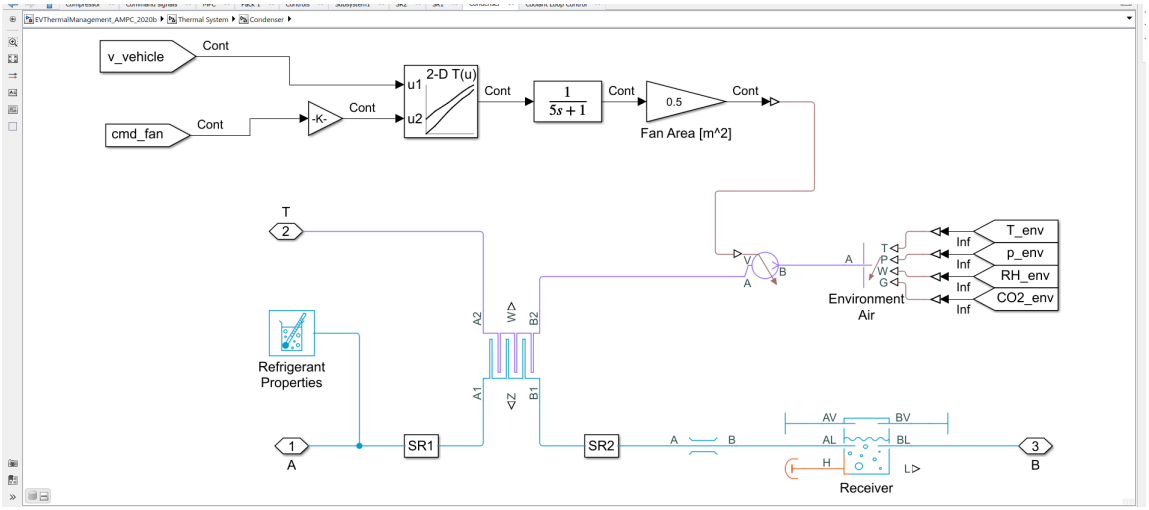


Figure 5: Fan Model Matlab 2020b

2.3 Controller

In the original model provided by Mathworks, a reactive control logic is used. Control systems are often reactive, which means that the Controller doesn't predict anything but reacts to an event that occurs (like the reaching of a certain threshold) by triggering actuators that change the state of the system after the feedback that come from the sensors. A straight example comes from the valves that were described before; these valves are actuated once a certain condition is reached. This is the easiest way to tune a Controller, but not the most effective one: the dynamic of a complex system can only be controlled effectively with a tool that keeps into account its complexity. That's why MPC becomes a valuable alternative. In the previous work, as mentioned, an **Adaptive Model Predictive Control** was implemented and is still the control system used in this model.

2.3.1 Model Predictive Control

Model Predictive Control (MPC) is a control scheme that is used to predict the behavior of the system over a finite time window, the horizon. At each time step, the optimal input is chosen based on a cost function that should be minimized. This is computed from the system states chosen in the formulation of the Controller. In a certain time instant, the sequence of optimal inputs is computed over a certain time horizon but only the first value for the actual timestep is picked, then the horizon is shifted. The MPC works with a closed-loop control system as can be seen in the picture below.[6]

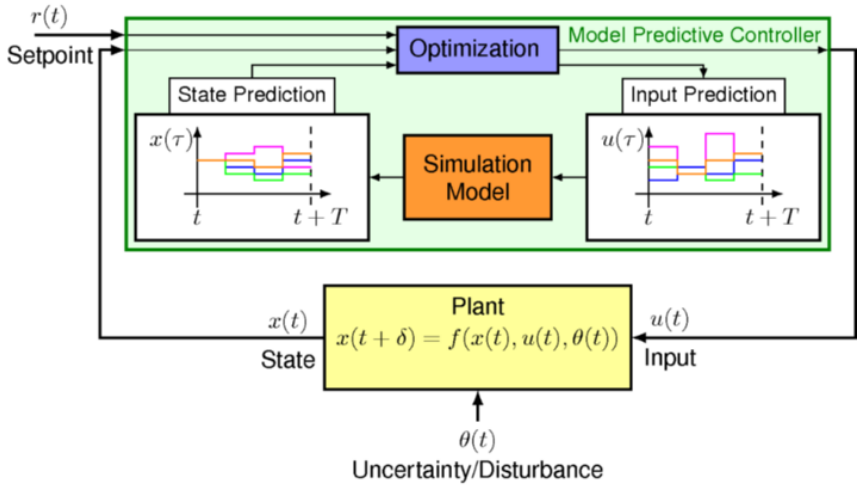


Figure 6: MPC scheme

In order to use an MPC in its basic formulation (for linear systems), the system to be controlled can be modeled as a **Linear Time-Invariant System (LTI)**, whose model can be implemented with State Space (SS) equations. These can be expressed both in continuous time and discrete time.

$$\begin{aligned} \dot{x}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned} \tag{2}$$

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{aligned} \tag{3}$$

The first equation explains how the state evolves in time as a function of the state and the input itself, while the latter explains how the chosen output depends on state and input. So, while **A** and **B** matrices are determined by the system structure and elements, **C** and **D** matrices are determined based on the chosen output.[7]
Given that, any LTI system can be expressed as a series of n state equations where n is the number of states that define the system, and these states are defined in terms of the matrices **A** and **B**, while the p outputs equations, where p is the number of outputs of the system, are expressed in terms of matrices **C** and **D**.

In some contexts (like this project), there are other variables to be considered, disturbances: these variables can be considered as inputs that are not controlled and that can be either measured or unmeasured. Following this statement, the final state equations (in discrete state) can be seen as:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu_u(k) + Bv_v(k) + Bd_d(k) \\ y(k) &= Cx(k) + Dv_v(k) + Bd_d(k) \end{aligned} \tag{4}$$

Where v are the measured disturbances and d are the unmeasured disturbances.[8]

Then, it is necessary to compute a suitable cost function for the linear (or linearized) system to give the proper importance to the key variables that influence the achievement of control objectives, distributing, at the same time, the appropriate weights to penalize commands or unwanted behaviors.
This cost is calculated through a certain prediction horizon, but the inputs are optimized only for the predetermined control horizon. After the final control horizon step, all the other inputs are considered constant, not optimized, until the end of the prediction horizon.

The standard cost function for the typical MPC can be formulated this way:[9]

$$J(z_k) = J_y(z_k) + J_u(z_k) + J_{\Delta u}(z_k) + J_\epsilon(z_k). \tag{5}$$

That essentially sums up different costs, each one of them linked to a different variable. Now each cost element can be summarized and explained.

$$J_y(z_k) = \sum_{j=1}^{n_y} \sum_{i=1}^{p} \left\{ \frac{w_{i,j}^y}{s_j^y} \left[r_j(k+i|k) - y_j(k+i|k) \right] \right\}^2. \tag{6}$$

The first equation says that for each output variable (from 1 to n_y), a summation that goes from the actual step to the end of the prediction horizon (from 1 to p) assigns a certain weight w, properly scaled by s factor since there are different physical entities in the sum of the costs, to the difference that arises between the reference value and the actual output. The summations are performed for the square of the weighted and scaled differences. This is done for each time step within the prediction horizon.

$$J_u(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^u}{s_j^u} \left[u_j(k+i|k) - u_{j,\text{target}}(k+i|k) \right] \right\}^2. \tag{7}$$

The second equation puts a weight on the input. The mechanism is completely similar to what has been explained before, but here, the difference between the actual input and the target one is evaluated.

$$J_{\Delta u}(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^{\Delta u}}{s_j^u} \left[u_j(k+i|k) - u_j(k+i-1|k) \right] \right\}^2. \tag{8}$$

The third equation penalizes the excessive variation of the input from one step to another. The proper setting of these equations depends obviously on the context and the wanted scope of the Controller

in the design phase.

It is worth adding that z_k is the vector of the decision variables of the optimization program (which is a Quadratic Optimization program); it is the vector of all the inputs within the prediction horizon (control horizon + constant input). In the end, in the actual step, only the first input is taken.

$$J_\varepsilon(z_k) = \rho_\varepsilon \varepsilon_k^2. \tag{9}$$

This last equation refers to the weight given to the constraints violation. There are indeed certain constraints that variables can't bypass, but with the slack variable ϵ_k these are taken less rigidly so that the system can operate also out of the constraints region, but with an increasing penalty to them associated.

2.3.2 Adaptive Model Predictive Control

MPC can give good results any time there is a linear system or a system that doesn't have great non linearity, so it can be used only for LTI systems, but if the Plant is strongly nonlinear or its characteristics vary consistently with time, LTI prediction accuracy might reduce and MPC would not be consistent for the application. **Adaptive MPC** can address this degradation by adapting the prediction model for changing operating conditions.

Moreover, the battery's system significant thermal inertia results in slow thermal response, necessitating longer prediction horizon for battery optimization.

AMPC overcomes the need to use a LTI system in the control logic by successive linearizations around the current system state for each timestep. After the system is linearized step by step, it operates as a standard MPC.

A nonlinear system of the type:

$$\begin{aligned} \dot{x}(t) &= f[x(t), u(t)] \\ y(t) &= h[x(t), u(t)] \end{aligned} \tag{10}$$

Can be linearized around the equilibrium state $\bar{x}$ corresponding to a constant input $\bar{u}$. [10]
In this context, small perturbations can be defined as:

$$\begin{aligned} \delta x(t) &= x(t) - \bar{x} \\ \delta u(t) &= u(t) - \bar{u} \\ \delta y(t) &= y(t) - h(\bar{x}, \bar{u}) \end{aligned} \tag{11}$$

Then, considering the Taylor expansions of first order of functions f (state) and h (output), it can be obtained that:

$$\dot{x}(t) \equiv f(\bar{x}, \bar{u}) + \left[\frac{\partial f}{\partial x}\right]_{(\bar{x},\bar{u})} \delta x + \left[\frac{\partial f}{\partial u}\right]_{(\bar{x},\bar{u})} \delta u \tag{12}$$

$$\delta\dot{x}(t) \equiv \left[\frac{\partial f}{\partial x}\right]_{(\bar{x},\bar{u})} \delta x + \left[\frac{\partial f}{\partial u}\right]_{(\bar{x},\bar{u})} \delta u \tag{13}$$

$$y(t) \equiv h(\bar{x}, \bar{u}) + \left[\frac{\partial h}{\partial x}\right]_{(\bar{x},\bar{u})} \delta x + \left[\frac{\partial h}{\partial u}\right]_{(\bar{x},\bar{u})} \delta u \tag{14}$$

$$\delta y(t) = y(t) - h(\bar{x}, \bar{u}) = \left[\frac{\partial h}{\partial x}\right]_{(\bar{x},\bar{u})} \delta x + \left[\frac{\partial h}{\partial u}\right]_{(\bar{x},\bar{u})} \delta u \tag{15}$$

Which means that the final linearized system, which can be considered an approximation of the nonlinear system in the neighborhood of the point($\bar{x}$,$\bar{u}$), has this form in its state-space representation:

$$\begin{aligned} \delta\dot{x}(t) &= A\delta x(t) + B\delta u(t) \\ \delta y(t) &= C\delta x(t) + D\delta u(t) \end{aligned} \tag{16}$$

Where the state matrices have been obtained by the Taylor expansions previously described.

2.3.3 AMPC in Simulink

It is now necessary to introduce how the AMPC works in the model used in this Master Thesis work. The AMPC, as said before, is here used to control the BTMS of the vehicle and it is implemented in Simulink through the **MPC Matlab Toolbox**.
Before implementing the toolbox, a formulation of the equations has been made, starting from a model without the cabin temperature regulation and, so, the HVAC system.
In this case, quite a simple formulation, to avoid unnecessary computational burden, has been realized. At first, it was hypothesized to control the compressor power by giving, at the same time, a certain weight to the Battery Temperature and to the State of Charge (SoC) of the battery, but the results were quite similar even without this contribution. So to derive the state-space model that describes the battery temperature dynamics, the first law of thermodynamics is used:

$$\dot{Q}_J - \dot{Q}_{BTM} = m_b c_p \dot{T}_b \tag{17}$$

Where $\dot{Q}_J$ is the heat generated due to Joule Effect, $\dot{Q}_{BTM}$ is the heat removed to cool the battery and m_b is the mass of the battery. To find a proper state equation for the controller, it it necessary to find how these terms are related to other variables, like disturbances.

The first equation that describes the rate of heat generation due to traction current, comes from the simple formula of the Joule effect, by considering a certain correcting coefficient (a) and a dependency of the resistance on both the temperature and the SoC of the battery. The rate of heat generation is calculated for all the cells (n_{cell}).

$$\dot{Q}_J = a \cdot n_{cell} \cdot R_0(T_b, SOC) \cdot i^2 \tag{18}$$

Then the rate of heat removal due to battery cooling is considered as a linear function of the power used by the compressor, whose coefficient of proportionality is expressed by a COP (a coefficient of performance

that has been calculated in the previous work) that is considered only a function of the environment temperature for sake of simplicity.

$$\dot{Q_{BTM}} = COP(T_{env}) \cdot P_{comp} \tag{19}$$

By what has been said before, the state equation at the basis of the MPC is therefore found.

$$\dot{T_b} = \frac{1}{m_b c_P} \left(a \cdot R_0(T_b, SOC) \cdot i^2 - COP(T_{env}) \cdot P_{comp} \right) \tag{20}$$

In this AMPC formulation the only state variable is the **Battery Temperature** T_b, the input of the Controller is the **Compressor Power** P_{comp}, which is the variable that is found through optimization, while there are only a few measured disturbances, the **Battery Current** i, and the **Environment Temperature** T_{env}. The output for this application has been considered the state itself.

The control subsystem of the compressor was implemented in Simulink from these considerations:

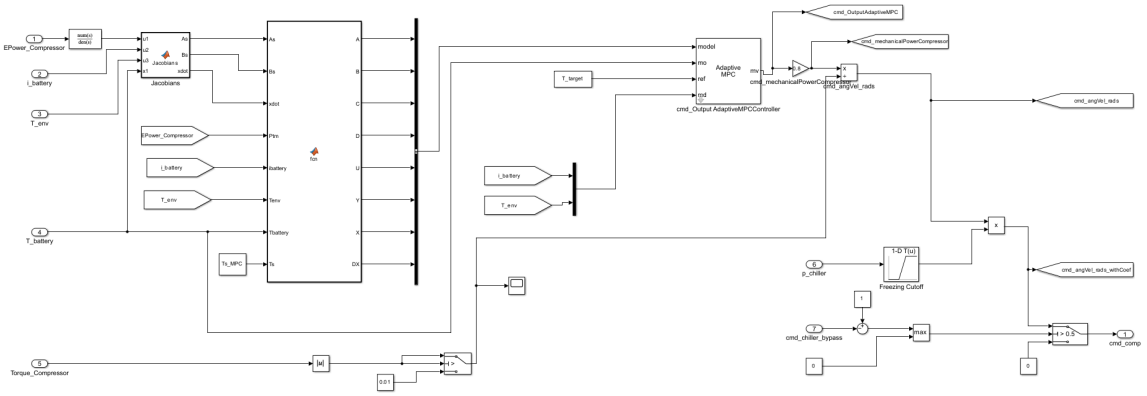


Figure 7: Controller: Simulink scheme

In this subsystem, the main block is the Adaptive MPC block, where the manipulated variable is optimized given certain inputs. Here, the manipulated variable is the power of the compressor, while the control input required by the component is the angular speed, easily

obtained from the torque data that comes from the Plant.
The AMPC Toolbox uses as inputs the linearized model, which, in this case, is updated at each timestep, the manipulated output (the battery temperature), the reference (the target temperature), and the disturbances before described. The linearized model is obtained by giving all the inputs (controlled and disturbances) to a Matlab function that generates the Jacobian matrices (Jacobians block), then with another Matlab function, these matrices are discretized for each timestep and ready to be given to the Toolbox as an LTI model description.

3 Real Time Simulation Settings

Real-time simulation consists of a simulation method for a system where simulation time and and physical time match; it is strictly correlated to the capacity of the system itself to respond to certain events within a certain timestep, and its implementation is paramount in a modern control system where, in every condition, the vehicle should be able to respond quickly to external sources of potential harm or abrupt condition changes, considering that in real-time environment 1 second of simulation corresponds to 1 real-time second. In this context, it is not only important how precise the dynamic representation of the system is, but also the time it takes to represent it properly.[11]

With the improved performances of the computational resources, it is possible to use tools and control systems that are more and more complicated and precise.
In order to perform a real-time simulation, different steps should be taken, but the main thing to do is make the system work in a fixed timestep. it is essential to make sure the right solver is chosen together with the correct timestep for a well-defined simulation.
Afterward, it is necessary to have the proper Hardware and Software to make it possible. In this thesis, as before mentioned, Hardware and Software from the dSPACE environment have been used.

3.1 Solver

By solver, we mean the mathematical tool that calculates the state variables of the model at every timestep during the simulation. Solvers solve, as the name suggests, the system of differential equations and, in the case of variable timestep, determine the next simulation timestep by taking into account system dynamics and error tolerances.

For this reason, variable timestep solvers are much preferred since they are more flexible and can adjust the interval between steps, while fixed timestep solvers have to follow a predetermined timestep for the

whole simulation time.
However, as mentioned before, for real-time simulation environments, only fixed timestep solvers can be used. Each one of them has its advantages and drawbacks, so whenever a simulation is started, it has to be chosen carefully to avoid errors.
It is indeed interesting to understand the solver used in this work and analyze why a certain solver comes to be much more stable than another one for a certain application.

3.1.1 Explicit vs Implicit Solvers

Explicit and implicit solvers exploit different ways of solving differential equations [12] and since their mathematical formulation is different, their efficiency and, as a consequence, their computational complexity, change accordingly. An explicit system can be expressed in the form:

$$\dot{x} = f(x) \tag{21}$$

This is called explicit since it is possible to calculate the derivative in every point by substituting x in the equation. At the same time, an implicit system can be expressed as:

$$F(\dot{x}, x) = 0 \tag{22}$$

Implicit solvers are usually better for dynamically stiff systems. A stiff system, as the one analyzed in this work, is more complex because it involves different dynamics: while a part of the system can vary slowly, another one can vary much more rapidly, so it requires very small timesteps; in this context, implicit solvers, even if much more heavy computationally, can deal more efficiently with oscillations.
While explicit solvers are quite slow for stiff problems, implicit solvers are better thanks to their mechanism of approximation of the next step solution, and are more mathematically stable for high timesteps, as it will be explained.
While the Controller can be dealt also with explicit solvers, the Plant

can only be solved with implicit solvers because it involves many different physical domains, with many different physical dynamics.

Few solvers were tried out during the tests, the more important ones were ode14x and ode1be, which turned out to be the most efficient. The choice of the solver, as mentioned before, is paramount to address the stiffness of a system.

3.1.2 ode14x: Implicit Extrapolation Solver

ode14x is a fixed time step solver that uses both Newton Method (that will be explained later for ode1be) and Extrapolation from the current state to compute model state as an implicit function of the state and the state derivative at the next time step. [13]
As formulated in Mathworks documentation, the equation at the basis of the solver is the following:

$$X_{n+1} - X_n - h\, dX_{n+1} = 0 \tag{23}$$

Where X is the state, dX is the state derivative, and h is the step size. Two parameters can be adjusted in Matlab: the number of Newton's method iterations and the order of extrapolation that can be chosen to compute the future state. The more iterations, and the higher the extrapolation order selected, the higher the accuracy, but, at the same time, the higher the computational burden. [14] Its formulation is very similar to ode1be, but the latter has a fixed number of Newton iterations and a fixed computational cost: this makes it much more suitable for a complex system like the one involved in this work.

3.1.3 ode1be: Backward or Implicit Euler Method

ode1be is a fixed timestep solver that is based on an ODE solver method called the **"Backward Euler"** method or **"Implicit Euler"**.[15] It is an implicit solver because, to find the future state, a function that is future state-dependent has to be used. As stated here:

$$\frac{y_{n+1} - y_n}{h} = f(t_{n+1}, y_{n+1}), \tag{24}$$

Where y_{n+1} is the future state of the system, y_n is the actual state and h is the chosen timestep.
This simple equation is given by the backward difference approximation shifted by a timestep:

$$\frac{dy}{dt} \approx \frac{y_n - y_{n-1}}{h} \tag{25}$$

The function must be transformed to make possible the resolution. By taking the equation (24) and, at first, moving the future state to the left-hand side and the actual state to the right-hand side, by subtracting from both sides the actual state, with a little mathematical manipulation the following equation can be found:

$$g(y_{n+1}) = y_{n+1} - hf(t_{n+1}, y_{n+1}) - y_n = 0 \tag{26}$$

The equation that comes out, $g(y_{n+1})$ can be highly non-linear and, for this reason, other numerical methods have to be used, like Newton's Method, because this becomes a root finding problem.
Newton's Method "is built around tangent lines. The main idea is that if x is sufficiently close to a root of f(x), then the tangent line to the graph at (x,f(x)) will cross the x-axis at a point closer to the root than x".[16]

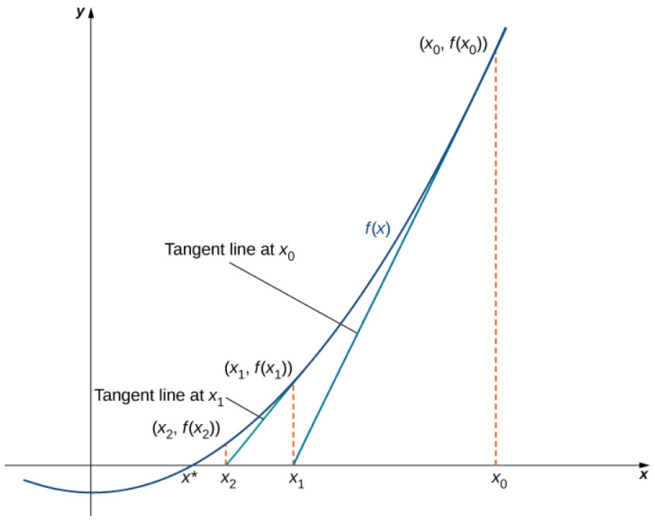


Figure 8: Newton's Method

Iteratively Newton's Method converges to the root.
So the formulation of the future state with Newton's Method can be found by the equation:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}. \tag{27}$$

Coming back to "Backward Euler", the formula can be expressed this way:

$$y_{n+1}^{(k+1)} = y_{n+1}^{(k)} - \frac{y_{n+1}^{(k)} - y_n - hf(y_{n+1}^{(k)}, t_{n+1})}{1 - h\frac{\partial f}{\partial y}(y_{n+1}^{(k)}, t_{n+1})}. \tag{28}$$

The "Backward Euler" algorithm for each timestep finds this way the slope of y and then moves forward by a timestep h. The found slope is used at the end of the actual interval.

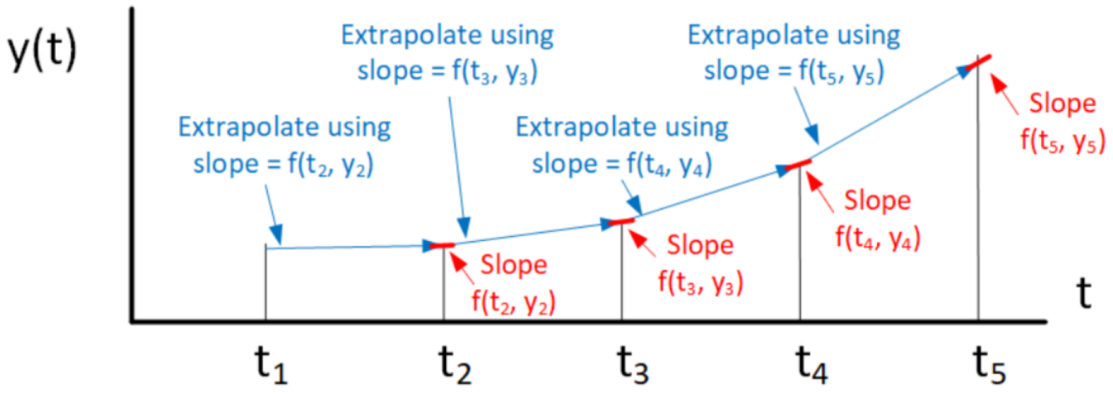


Figure 9: Backward Euler example

Backward Euler happens to be a very stable method even for high timesteps, and its stability is very much suitable for this application; indeed while the dynamic of the compressor is quite rapid, with very steep variations when its activation is demanded by the control system, at the same time, the battery system's (and all the other components) significant thermal inertia results in slow thermal response, and this dynamic is quite different from the one of the compressor as before said. Furthermore, there are many electrical components with a much faster dynamic.
This makes the system particularly stiff, so it is quite more difficult to solve it with high timesteps, as the one used in this research. This makes ode1be particularly suitable for the model due to its stability, especially for high timesteps.

The stability of the "Backward Euler" method can be analyzed starting from the exponential growth ODE:

$$\frac{dy}{dt} = \lambda y \tag{29}$$

Then by discretizing and manipulating the equation it can be found that:

$$y_{n+1} - h\lambda y_{n+1} - y_n = 0 \tag{30}$$

Since this is a linear equation, it is extremely easy to find the value of the future step:

$$y_{n+1} = \frac{y_n}{1 - h\lambda} \tag{31}$$

So intuitively the formulation of the exact solution has this form:

$$y_n = \left(\frac{1}{1 - h\lambda}\right)^n y_0 \tag{32}$$

Then, to analyze the stability of the system, the domain of the possible solutions of λ is restricted to the negative field, so that the exact solution decays to 0 when t tends to infinity. At the same time, the numerical solution should follow the same pattern, i.e. decay to 0 as the exact solution, when t tends to infinity. So by considering the stability condition:

$$\left|\frac{1}{1 - h\lambda}\right| < 1 \tag{33}$$

Considering that lambda is often a Real number, the condition for stability is:

$$|1 - h\lambda| > 1 \tag{34}$$

And since h is always positive because it is the timestep, and λ is always negative as stated before, this is always true. This makes Backward Euler stable even for high timestep values as the one chosen.[17]

3.2 dSPACE Hardware

3.2.1 Scalexio

For this application, two different devices are used.[18]
The first Hardware used is **dSPACE Scalexio LabBox** in its 19-slot chassis variant.
Scalexio LabBox is a modular and scalable real-time system designed for **Hardware-in-the-loop (HiL)** simulations and Rapid Control

Prototyping (RCP) in various application fields like automotive, aerospace, and industrial automation.

The presence of the slots makes the system flexible for different uses; slots can be added and removed according to the needs. So, not only it is equipped with a multi-core processor, but it can be upgraded with additional boards due to its modularity.
Moreover, it is a low-noise system, which is optimal for laboratory activities. It can be used both as a stand-alone real-time system and as an I/O carrier for other boards.

It uses high-performance processors, often powered by multi-core processors, to handle real-time simulations and control systems. Moreover, the modularity allows for upgrading the computational power by adding other processing boards. Scalexio high-performance processors are used in order to simulate the Plant of the thermal system, which in this case study results in a highly demanding computational system.

Figure 10: dSPACE Scalexio LabBox Hardware

Scalexio LabBox has several interfaces that are used in the auto-

motive field:

- CAN (Controller Area Network) - 6341 Board
- Ethernet
- LIN (Local Interconnected Network)
- FlexRay
- SPI, I2C, UART, and others.

While some Scalexio LabBox are equipped with a LIN board it is not the case with this Hardware. Anyway, the modularity of the Hardware makes possible the extension to other boards.

Moreover, other I/O boards provide I/O analog and digital channels for control systems:

- Analog Input/Output modules with high-resolution ADC/DAC.
- Digital Input/Output modules for precise signal generation and capture.
- PWM, Encoder, and Resolver inputs for motor and actuator control.

Scalexio Hardware is compatible with the Matlab/Simulink environment, which makes possible the use of their product models for this work goals, and is compatible with all the Software developed by the dSPACE environment, like, in this case, ConfigurationDesk and ControlDesk.

Here are the technical characteristics of Scalexio LabBox:

Parameter	Specification
SCALEXIO LabBox	SCALEXIO LabBox 8-Slot
General	• Chassis with 8 slots • 7 slots for SCALEXIO I/O boards • 5 extended I/O slots[1)] • 1 system slot reserved for a DS6001 Processor Board or DS6051 IOCNET Router
	• Temperature-controlled active cooling • I/O boards software-configurable via ConfigurationDesk • Board exchange via front flap using an ejection lever • Status LED and Kensington® Security Slot
Ambient temperature	• 0 °C ... 50 °C (32 °F ... 122 °F)
Operating humidity	• 5% ... 95% (non-condensing environment)
Size (width x height x depth)	• Desktop version: 224 x 193 x 394 mm (8.8 x 7.6 x 15.5. in) • Rack-mount version: 483 x 178 x 355 mm (19 x 7 x 14 in) • Covered rack-mount version: 483 mm x 178 mm x 493 mm (19 in x 7 in x 19.4 in)
Mass	7.25 kg (without boards)
Power supply	100 ... 240 V AC, 50 ... 60 Hz; 350 W

Figure 11: dSPACE Scalexio LabBox Technical Characteristics

3.2.2 MicroAutobox III

The **MicroAutobox III** is a compact Rapid Control Prototyping (RCP) unit that can replace an Electronic Control Unit (ECU) in a vehicle or control system, allowing the user to experience and test control functions in a real environment. It is indeed compliant with automotive standards on shock and vibrations. The control Software is developed model-based using Matlab/Simulink. This Hardware is indeed compatible with Mathworks environment and with dSPACE environment Software like ConfigurationDesk and ControlDesk.
The MicroAutobox III can be equipped with I/O interfaces as well as

bus and network interfaces for various applications.
The MicroAutobox III used in this work is equipped with a DS 1511 Multi I/O board.

It supports a wide range of standard automotive and industrial communication protocols:

- CAN (Controller Area Network)
- Ethernet
- LIN (Local Interconnected Network)
- FlexRay
- XCP on Ethernet, others...

Moreover, it offers also a comprehensive range of analog and digital I/O to interface with sensors, actuators, and others:

- Analog I/O: 16-bit resolution for high-precision analog signal acquisition and generation.
- Digital I/O: configurable as standard digital I/O or pulse width modulation (PWM) for controlling actuators.
- PWM and Frequency I/O: for controlling motors and other pulse-based systems.
- Encoder Inputs: for interfacing with rotational devices (e.g. motors and sensors).

MicroAutobox III is characterized by lower processing power compared to Scalexio LabBox, but it is ideal for real-time applications since it is designed to represent real-world conditions. For this reason, it is used to simulate the Controller, which is much lighter than the full thermal model uploaded to Scalexio LabBox.

Figure 12: dSPACE MicroAutobox III Hardware

3.3 dSPACE Software

3.3.1 ConfigurationDesk

ConfigurationDesk is an intuitive interface to set up the model for the real-time simulation. it is widely used for both Hardware-in-the-loop (HiL) tests and rapid control prototyping (RCP) applications. It configures real-time Hardware and the connected behavioral models.
ConfigurationDesk facilitates smooth interaction with Simulink, which allows the user to jump from one tool to the other easily.
To connect any electrical device, ConfigurationDesk provides many I/O function types, which are separated from the behavioral model, so they are flexible and reusable.
It allows an intuitive configuration of the I/O ports and it makes it possible to automatically generate the C code that is then to be loaded on dSPACE Hardware targets.
In this work, it has been used to initialize the simulations, to configure

the CAN communication in the double Hardware configuration, and, of course, to build the C code for the online applications.

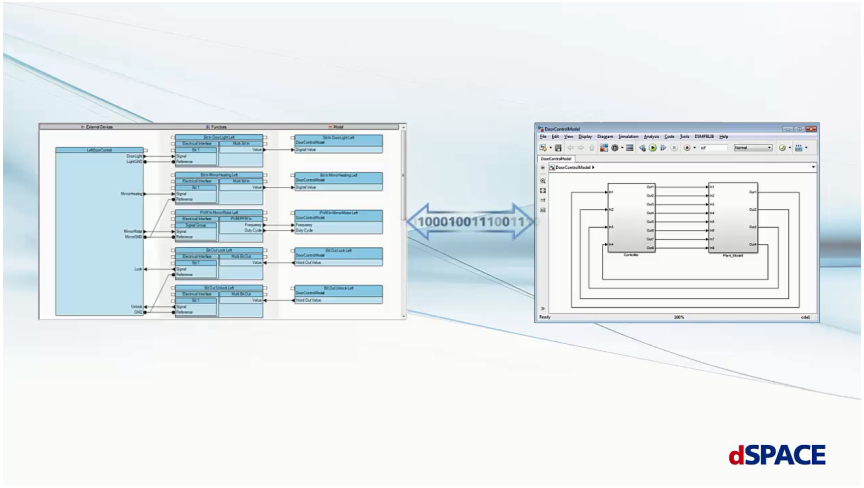


Figure 13: ConfigurationDesk

3.3.2 ControlDesk

ControlDesk is a universal, modular experiment and instrumentation Software for ECU development and data analysis. ControlDesk can access virtual ECUs generated with dSPACE's ConfigurationDesk and Simulink models that are simulated offline on the PC.
It can be used for Rapid Control Prototyping (RCP), Hardware in the Loop (HiL) simulations, Electronic Control Units (ECUs) measurements, calibration, and diagnostics. Moreover, it allows access to system buses such as CAN, CAN FD, LIN and Ethernet.
In this thesis, the Software has been used mainly to have a graphical interface, to monitor all the data that are considered useful for the overall analysis and, moreover, it can be used also to monitor the data that are exchanged through busses, like CAN bus in this case.
It is compatible with the Matlab/Simulink environment, and real-time application of models to them related are here analyzed after being configured in ConfigurationDesk.

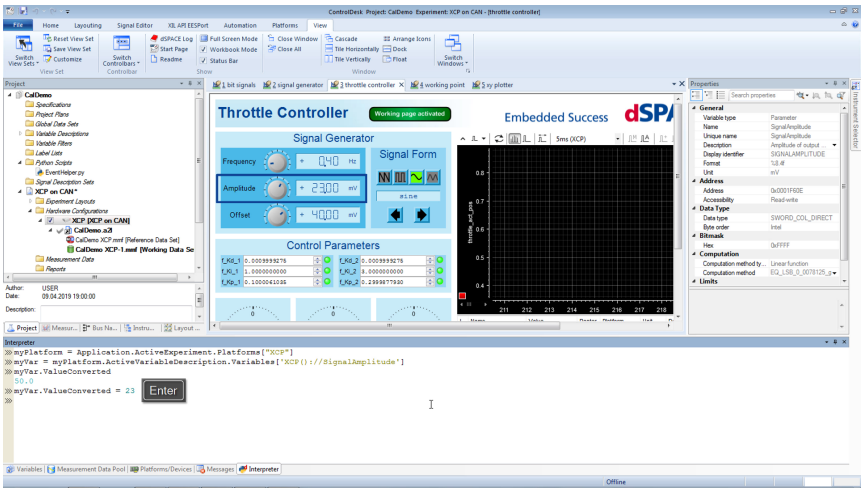


Figure 14: ControlDesk

4 Single and Double Model Simulation

4.1 Simulation Scenario

The **initial conditions** for this first simulation are the ones that were optimized in the previous work. The initial temperature for all the simulations is 26°C. The other parameters are never changed. The simulations are done only with this setting to understand their feasibility in real-time.

```
%% MPC parameter

%% Time step
% MPC control time step
Ts_MPC = 1.0;

%% Scale
% Scale factors
T_batt_scale = 10; % span of battery temperature in °C (1)
current_scale = 100; % span of battery current in A (200)
T_env_scale = 10; % span of enviroment temperature in °C (21)

%% Objective function
% Objective function weights
WeightTbattery = 1;
WeightSOC = 0;
WeightEPowerComp = 0.1;
WeightEPowerRateComp = 0.5;
WeightECR = 100000;

%% Prediction and control horizons
% Prediction horizon
p = 9;
% Control horizon
pc = 4;

%% Constraints
% Battery temperature limits
min_batt_T = 25;
max_batt_T = 35;

% Power compressor limit
MaxEPowerCompressor = 6000; % W
% Rate of change of compressor power ( to avoid unrealistic changes of compressor power)
Pcomp_RateMin = -1000*Ts_MPC; % minimum W/step that can change the compressor power
Pcomp_RateMax = +1000*Ts_MPC; % maximum W/step that can change the compressor power
```

Figure 15: Simulation Scenario: MPC parameters

About **MPC parameters**, everything was kept as in the previous work, except for the weight given to the rate of change of the compressor power, which is set to 0.5. This is done so that the compressor dynamic is damped: with this parameter change, the compressor be-

havior is still quite the same, but the output power has a smoother rate of change, which helps the solver to deal with different types of dynamics without giving errors as output.
The result in terms of absolute power is very similar but with fewer oscillations.

For all the simulations, **ode1be** is used, even when Plant and Controller are decoupled.
Since the full system works with ode1be and the reference results obtained in the offline simulation are given by Backward Euler, the controller's solver isn't changed because the main goal is to assess the differences with the offline simulation due to the real-time constraints; another solver would add other errors that could lead to other interpretations.
Anyways, for the double model simulation, an explicit solver will be compared to the implicit one, to assess differences and possible future developments.

4.2 Single Model Simulation

The first simulations are carried out in the dSPACE environment by only using Scalexio LabBox as Hardware: the idea is to test at first the full model and its performance in real-time in order to understand the feasibility of the simulation.
All the necessary steps will be now explained.

At first, the code is simulated on a Matlab/Simulink environment with the proper settings adjustments to make it work on the dSPACE environment. It is necessary to go to Simulink Settings and adjust the parameters relative to the solver (ode1be) and the timestep. Although different timestep configurations have been tried out, in the end, the one used is 0.2 seconds. This turns out to be the best timestep possible for the simulation. The choice is done considering that 0.2 seconds is the highest possible for this model and the solver to have a finite and errorless simulation. This timestep is anyway much lower than the

MPC one, which is as high as 1 second.

In "Code Generation" it is necessary to use dsrt.tlc (dSPACE Run-Time Target for VEOS on Windows and ConfigurationDesk). This allows selecting the correct system target file to which the code should be generated.

Then, after the Simulink Model is ready and the parameters have been adjusted, ConfigurationDesk can be opened. In the "New" window, the root of the file is selected together with the name of the project and the application.
In ConfigurationDesk, several applications can be initialized within the same project, and this function is quite useful in the double model simulations.
Then the Simulink model(s) that will be part of the project can be selected (in the picture, the case of the double model scenario can be seen) together with the relative Hardware.

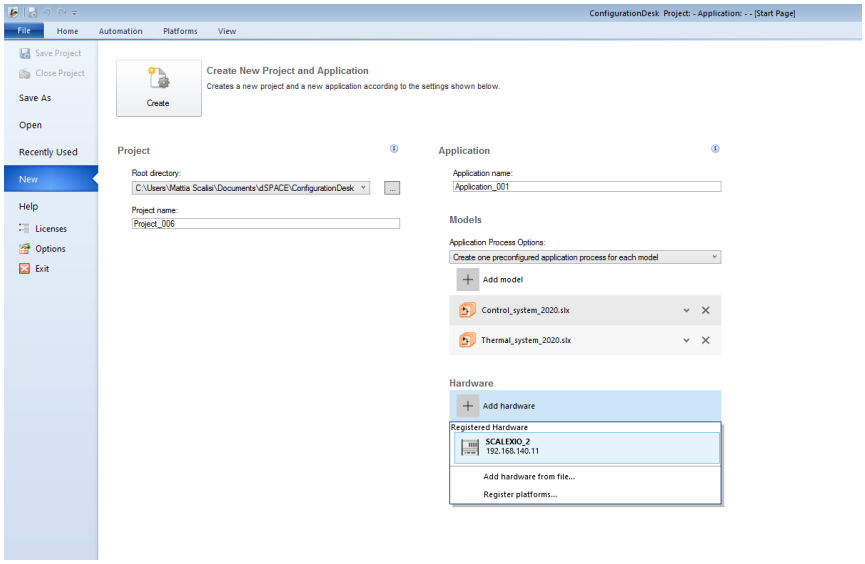


Figure 16: ConfigurationDesk: New Project

Inside ConfigurationDesk there are different windows with different functions. In the "Model Function" window, there is an overview of the

main function blocks that are uploaded into dSPACE from Simulink.

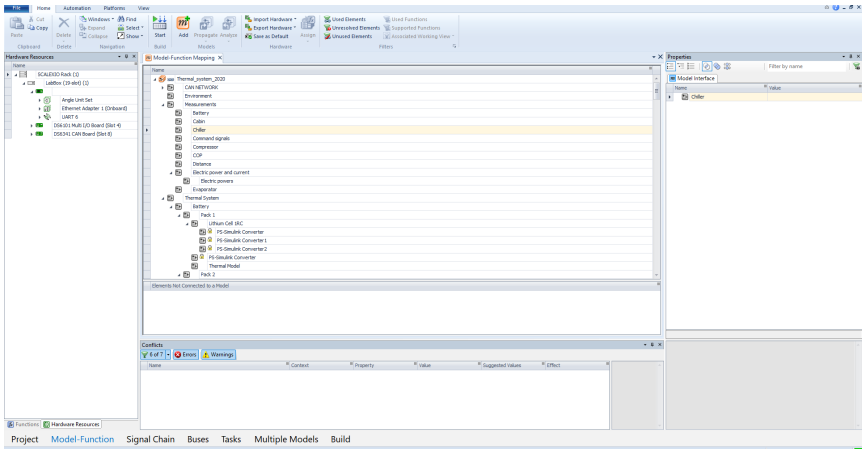


Figure 17: ConfigurationDesk: Model Function

Then, for the first run with the full system working on dSPACE Scalexio, there are only a few other passages that should be made. At first, proper simulation parameters should be assigned in "Tasks". The parameters to adjust are **overrun** count and **stack size**. For the full system, the count of the overruns had to be increased to make the simulation continue to run in real-time.
Each Task has to work within a given timestep: there is a factor called the **Real Time Factor** [19] that has to be lower than 1 and it is expressed as:

$$\text{RTF} = \frac{\text{simulation time}}{\text{clock time}} \tag{35}$$

Whenever the idle time, which is the spare time needed to process data and communicate within the I/O environment, isn't enough, overruns occur. These always lead to some kind of inaccuracy. When this happens, the system can't always schedule the next timestep operations and could go directly to the following one, avoiding some crucial operations.

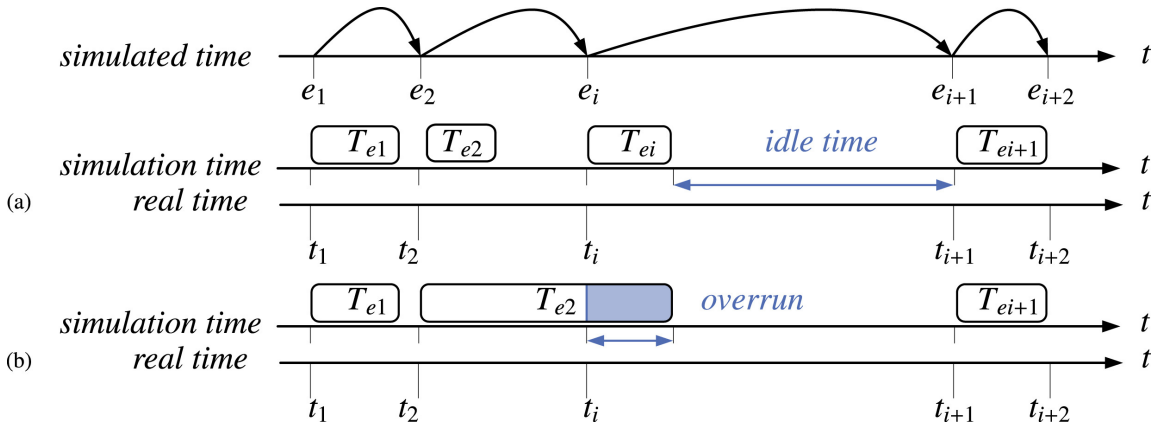


Figure 18: Real-time overruns

Anyway, given the fact that the Simscape Plant is very complex and uses, in its real-time simulation, a much lower timestep than the one used by the MPC, the stability of the simulation isn't affected. Moreover, these events happen only a few times during the simulation, primarily along faster dynamic changes, such as when the compressor is turned on: as it will be shown later, results are not of much concern. If the system hadn't accepted overruns, errors would have been present, and the simulation wouldn't have started.

Furthermore, the full system requires a minimum stack memory size higher than the custom one of 128kB. Therefore, it was increased up to 2048 kB. In the dSPACE environment, each periodic task can be given its memory allocation for efficiency's sake. Every multi-core processor, such as Scalexio LabBox, has a Memory Stack for each thread, and its role is to store the local variables relative to C codes executed within that thread; during the processor operations it works dynamically with data with a LIFO logic (Last In - First Out logic).[20]

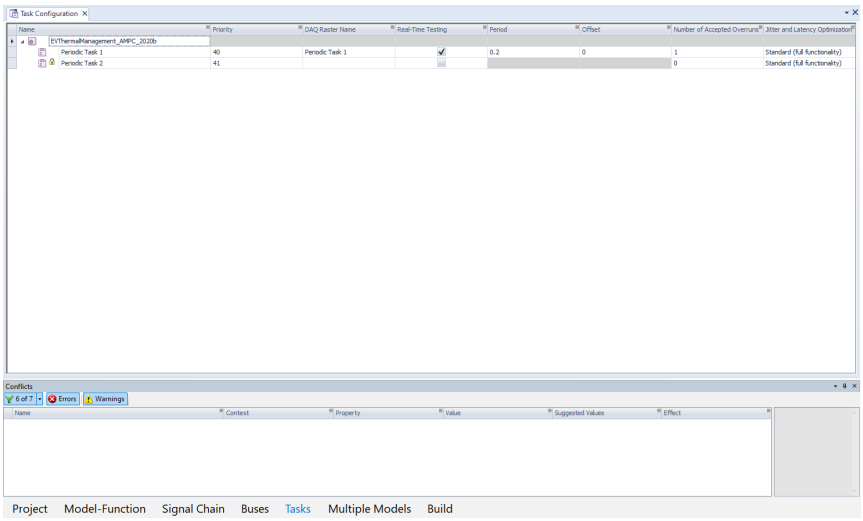


Figure 19: ConfigurationDesk: Tasks

Here setting "Number of Accepted overruns" at 1 was acceptable; it should be remembered that the goal of these simulations is not to prove that the Controller works for all the conditions, so this parameter is quite specific to the actual simulation. Eventually, it can be set to 5 so that the system is errorless in all the conditions. It has to be mentioned that this is not the number of overruns, but the maximum acceptable number per simulation step: in the dSPACE environment, there is a certain tolerance for a given amount of overruns at each timestep: the system continues to work in real-time but with more inaccuracies.
Then into the "Build" window, pressing the "Build" button generates a code in C language of the Simulink model to load on the Hardware and make it run in real time. After the C code is generated, a SDF file is given and ControlDesk can be opened. Before building up the code, in "Build Settings" it is possible to deactivate "start Real-Time simulation" so that the simulation is started only when everything is ready on ControlDesk, and not right after the code is uploaded on the Hardware.

When ControlDesk is opened, "New Project + Experiment" has to be selected.

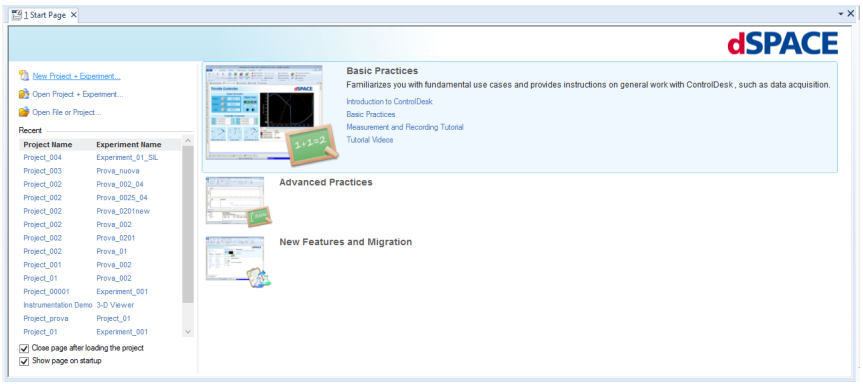


Figure 20: ControlDesk: New Project

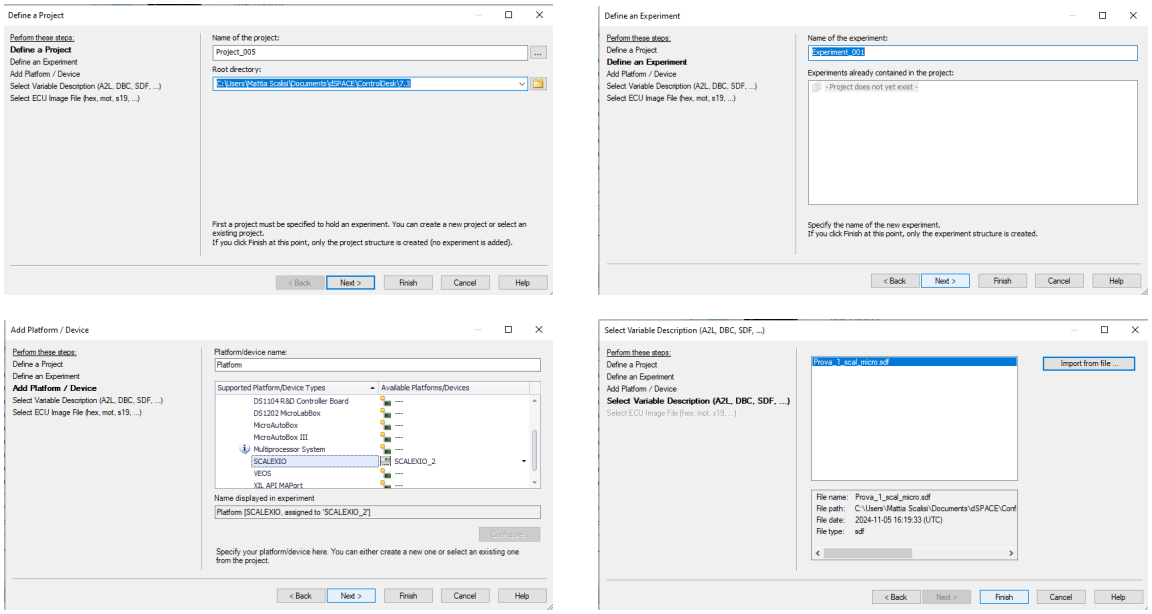


Figure 21: ControlDesk: Hardware and Variables Description

After defining a project and an experiment name, the platform relative to the project has to be chosen (Scalexio in this case) and the description of the model that was generated through the build button in ConfigurationDesk is uploaded (the SDF file).

After ControlDesk is opened, it can be seen that the model, which has been already uploaded, can be sent online. But before that, a new

layout for the given experiment has to be made. There are various instruments to be added and the ones used the most in this work are the "time plotter", which, as the name suggests, displays a given variable as a function of time (variables can be added with the right click on the mouse) and the "display", which as the name suggests, displays the actual value of the signal.

After the signals are set, the system can be sent online. A possibility is to press start recording, so that ControlDesk starts recording the values of the chosen signals right away. If the online button is pressed before, the system starts doing all the required real-time calculations without recording the signals, so in the postprocessing phase, this can lead to time-aligning problems.

After the recording is done and ready, it can be converted to a .mat file that can be used in postprocessing.

4.2.1 Results

To analyze the results, 4 different types of errors have been used to provide a complete picture of the differences between the simulations.[21]

- **Root Mean Square Error:** it is the square root of the mean squared errors. This can give a straightforward idea of the entity of the error, being not normalized, and, speaking of a squared error, it is more sensible to outliers and peaks, but at the same time, takes into account the direction of the error.
- **Mean Absolute Error:** it is the mean value of the errors, so it doesn't keep into account the direction of the error, but, as its main advantage, it is less influenced by peaks and outliers, so it is more robust.
- **Normalized Mean Absolute Error:** This is the MAE, normalized for the mean value, being expressed in percentage, useful to

make a comparison between parameters with different scales. In this case, the error was normalized excluding the 0 values of the compressor's parameters: the mean value would have been too low, generating an unreasonable error because the compressor operating window is reduced within the driving cycle.

- **Max Absolute Error:** it is the maximum absolute error between the two simulations.

These errors are related to the 3 parameters that have been analyzed, the ones related to the compressor, **Electric Power**(W), and **Command** (rad/s, as calculated by the controller, not yet saturated to the maximum compressor speed) and **Target Temperature** (°C), the objective of the control.

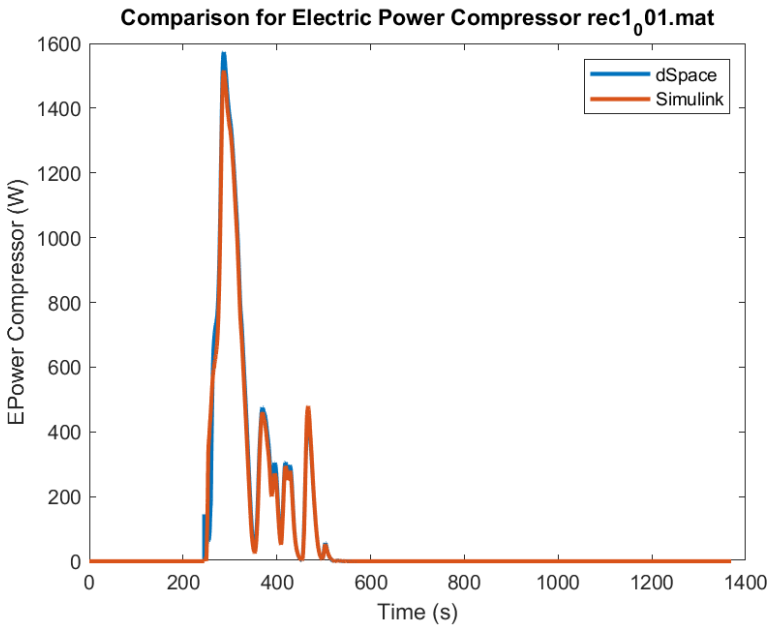


Figure 22: Single Model Simulation: EPower Compressor

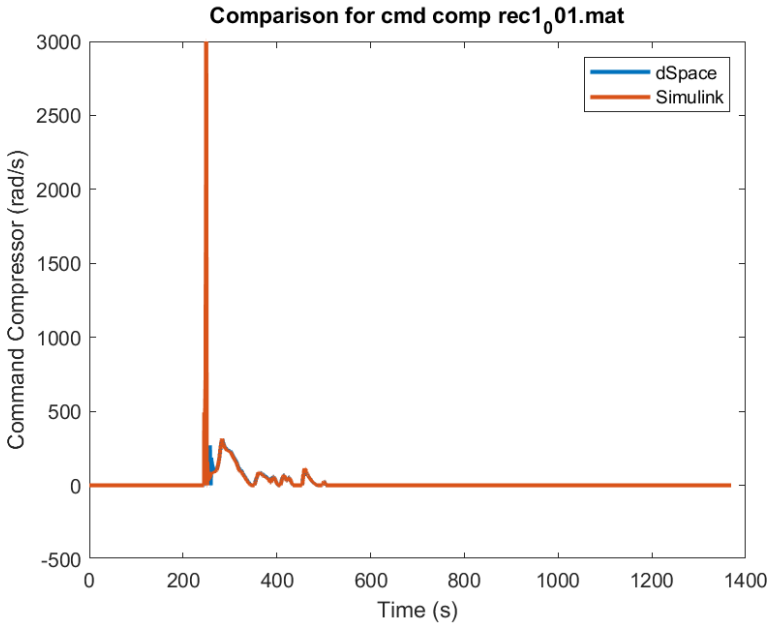


Figure 23: Single Model Simulation: Command Compressor

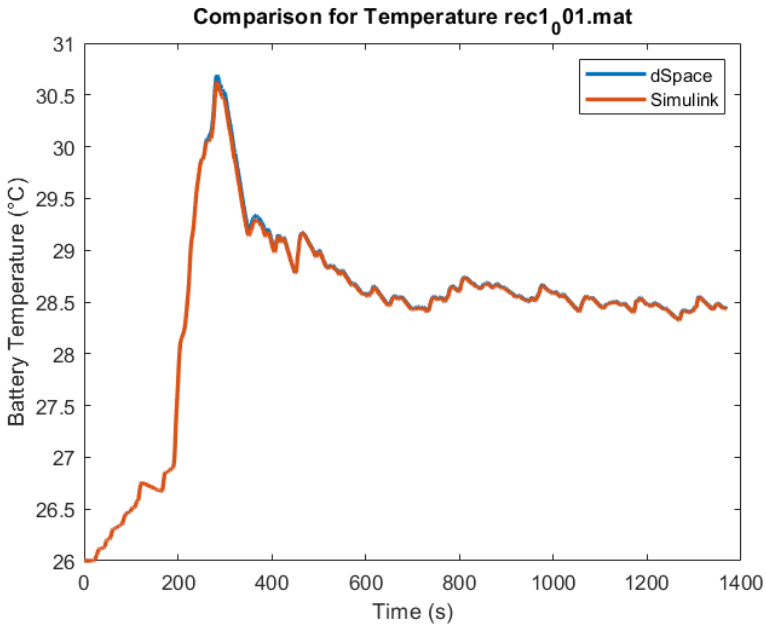


Figure 24: Single Model Simulation: Temperature

Parametro	EPower Compressor	Command Compressor	Temperature
RMSE	25.82 W	87.17 rad/s	0.019 °C
MAE	6.39 W	5.38 rad/s	0.011 °C
NMAE	2.80%	5.12%	0.04%
MAXAE	318.88 W	2966.23 rad/s	0.073 °C

Table 1: Results Single Model

In this first simulation, it can be noted that the results in real-time differ from the ones in Simulink, especially when the compressor is turned on.
This can be justified by the fact that when this happens the Compressor Command has some important oscillations and peaks that can lead to errors, as can be seen in Figure 23.
The **RMSE** are quite high, especially for the first two parameters due to the scale of the values for the Electric Power and due to the initial peaks and oscillations for the Compressor Command.
The **MAE** are much lower and their values are acceptable considering the scale of the variables while the **NMAE** is higher for the Compressor Command: the peaks are penalized in a normalized error to the mean value (even if 0 aren't considered). Anyway, considering the initial oscillations, it is a reasonably low error.
Moreover, peaks relative to the command of the compressor are that high in absolute terms also because the parameter isn't saturated to the maximum compressor speed: the ideal command requested by the controller is here considered.
Anyways, the Temperature, which is the object of control, has very satisfactory error values, so the Controller keeps up well, balancing the initial offset along the way to reach the **Target Temperature** as shown in Figure 24.
The oscillations are probably given because the whole system (Controller + Plant) is simulated in a single memory stack, leading to some inaccuracies in the data when the computational load is higher and, at the same time, the highest error happens whenever some overrun occurs as expected.

4.3 Double Model Simulation

With double model simulation, the system is tested by dividing Controller and Plant into two different tasks when uploaded on Scalexio LabBox. So, for this sake, two different Matlab models are created, one for the Controller and the other for the Plant. More specifically, this is the configuration of the **Thermal Plant** model:

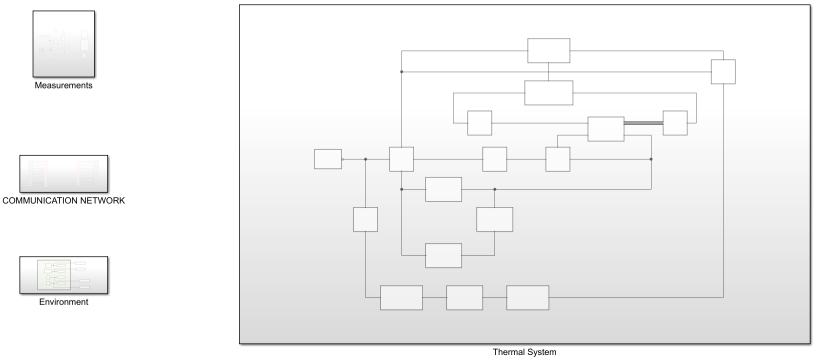


Figure 25: Thermal Plant

While this is the configuration of the **Controller** model.

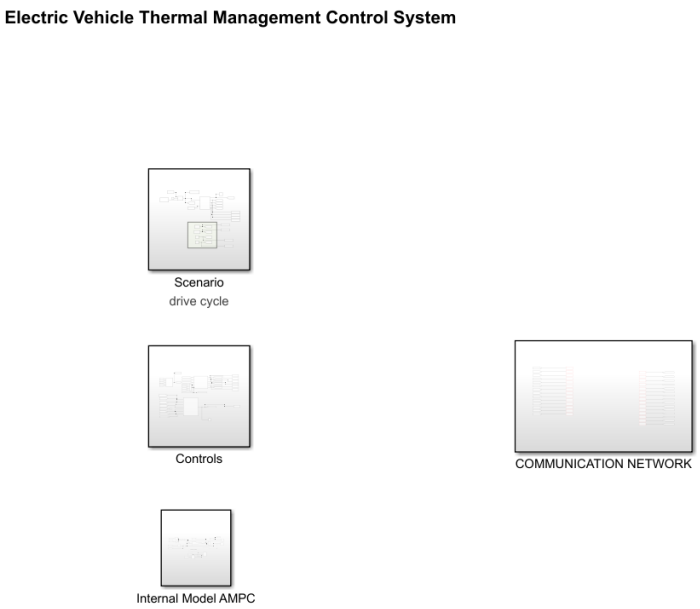


Figure 26: Controller

Concerning the original model, the "Scenario" subsystem can be found in both models to avoid the exchange of an excessive number of signals, while the "Measurements" subsystem is kept in the "Plant" model for the very same reason since its variables are mainly used in the Thermal system.
An additional subsystem is then added to initialize the communication between the two models on dSPACE. So a new subsystem, "Communication Network", has been created.

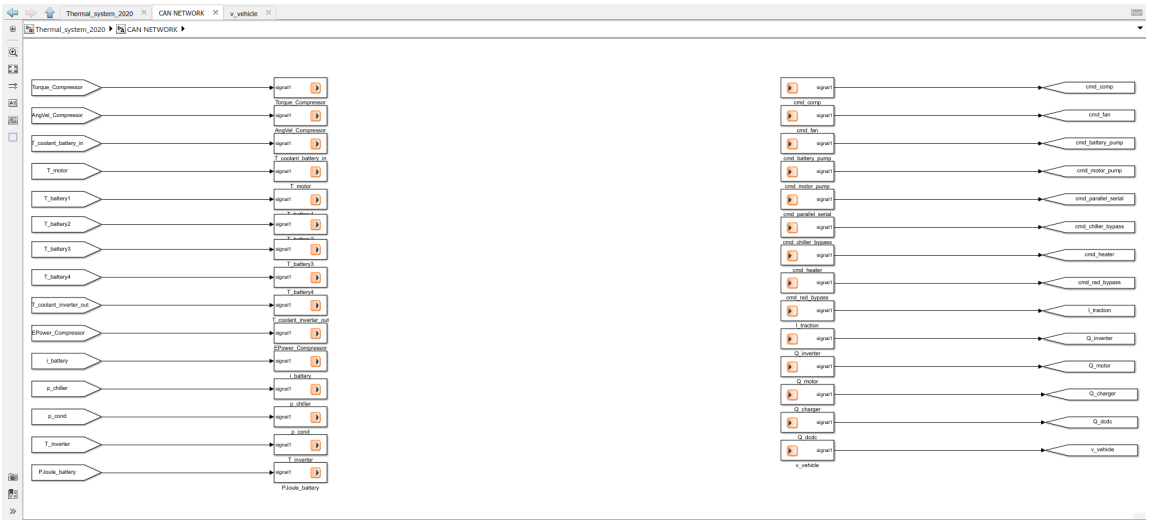


Figure 27: Communication Interface

In this specific case, it can be seen that two dSPACE blocks available in the Matlab environment are used: data inport and data outport blocks are the proper way to make the two models communicate with each other in ConfigurationDesk, and are native to the dSPACE environment. The Communication Networks are the same and opposite in terms of sent and received signals.

Once ConfigurationDesk is opened, as it was previously shown, the two models are opened and uploaded in a single unit, which is, also in this case, Scalexio LabBox. The passages are the same as before presented, but, this time, before building, it is necessary to go to

the "Multiple Models" window. Here, there are the signals from the Simulink models that have to be connected reciprocally.
After the connections are complete, the button "Propagate" can be pressed: this makes the changes available and observable in the Simulink models.

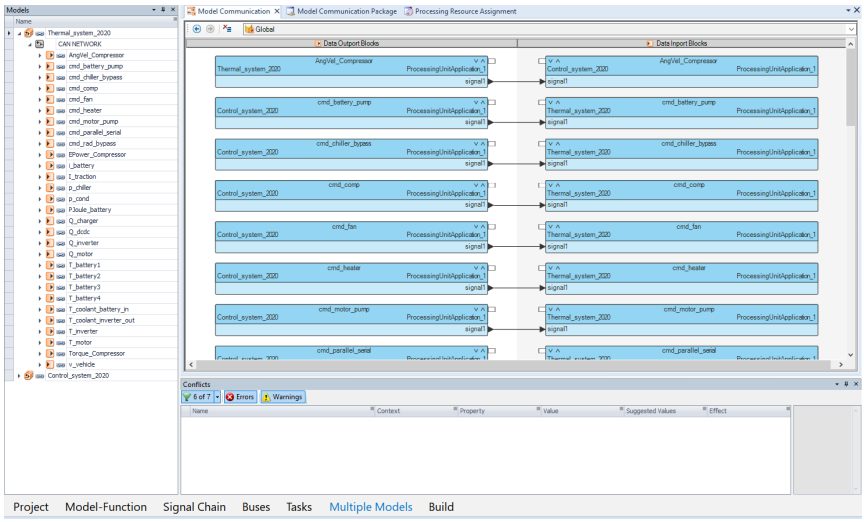


Figure 28: ConfigurationDesk: Multiple Models

This time there are two different tasks to set, and it can be noted that the Controller here can work with a very low timestep (Figure 29). This is, anyway, unnecessary, since the Controller works with a time-step equal to 1 second: using a much lower timestep doesn't change how the Controller operates, so results would be the same; in the end, 0.2 seconds was used for both tasks.
It can be observed that also in this case, allowing a certain number of overruns is necessary to complete the simulation in time.
Then, for what concerns the other passages, they are very similar to the ones that have been shown before.

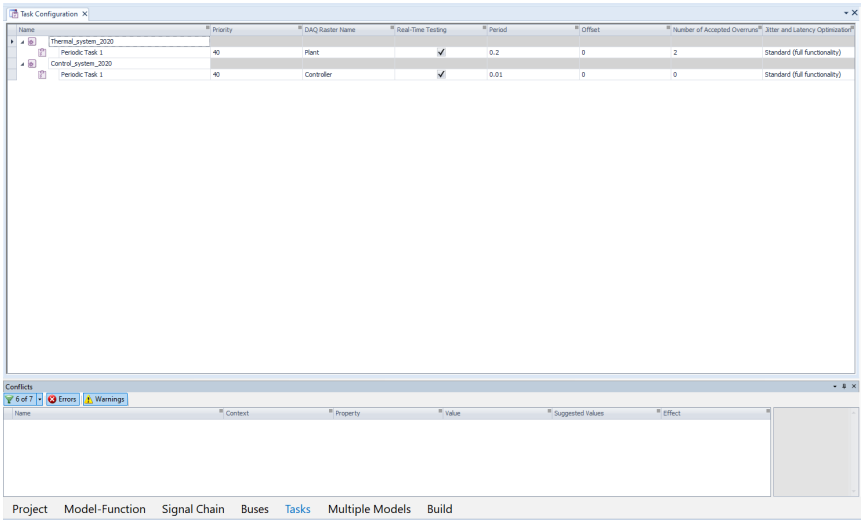


Figure 29: ConfigurationDesk: Tasks Double Model

4.3.1 Results

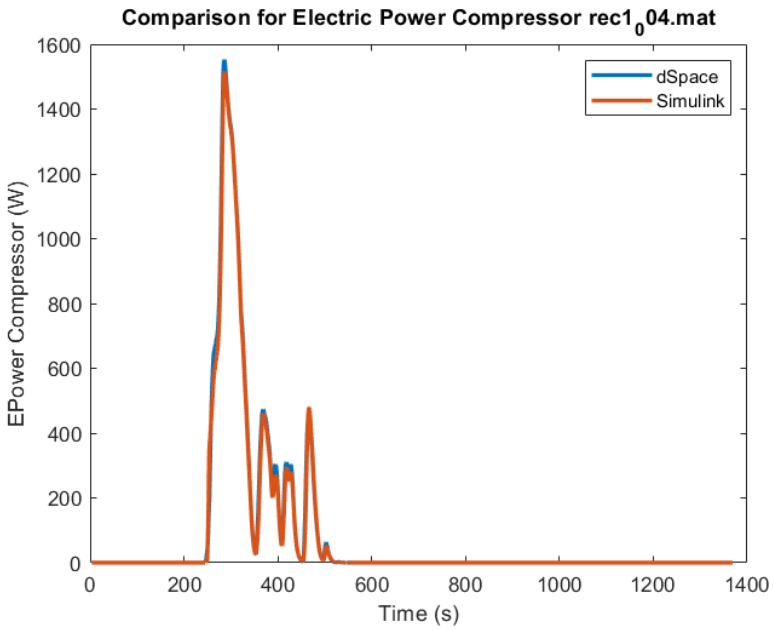


Figure 30: Double Model Simulation ode4: EPower Compressor

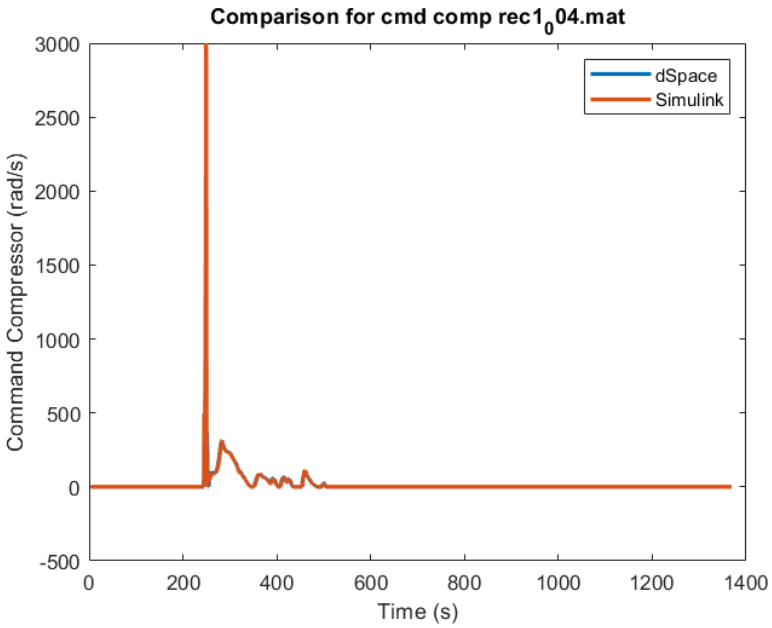


Figure 31: Double Model Simulation ode4: Command Compressor

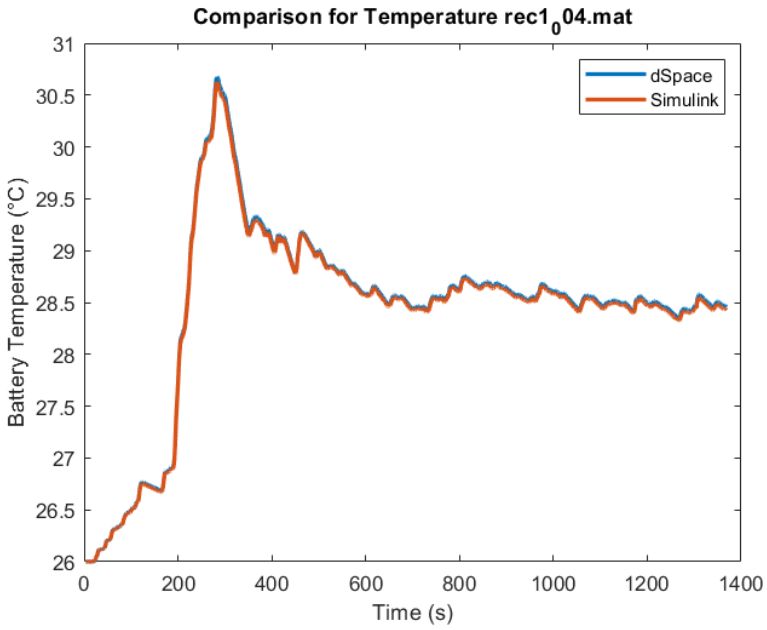


Figure 32: Double Model Simulation ode4: Temperature

Parametro	EPower Compressor	Command Compressor	Temperature
RMSE	13.64 W	76.68 rad/s	0.023 °C
MAE	3.86 W	4.57 rad/s	0.021 °C
NMAE	1.69%	4.31%	0.07%
MAXAE	165.41 W	2711.73 rad/s	0.054 °C

Table 2: Results Double Model ode4

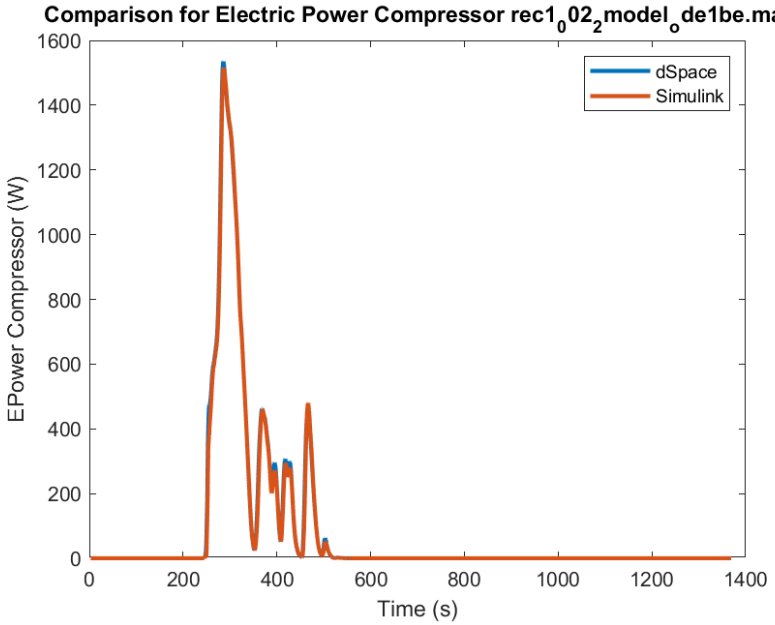


Figure 33: Double Model Simulation ode1be: EPower Compressor

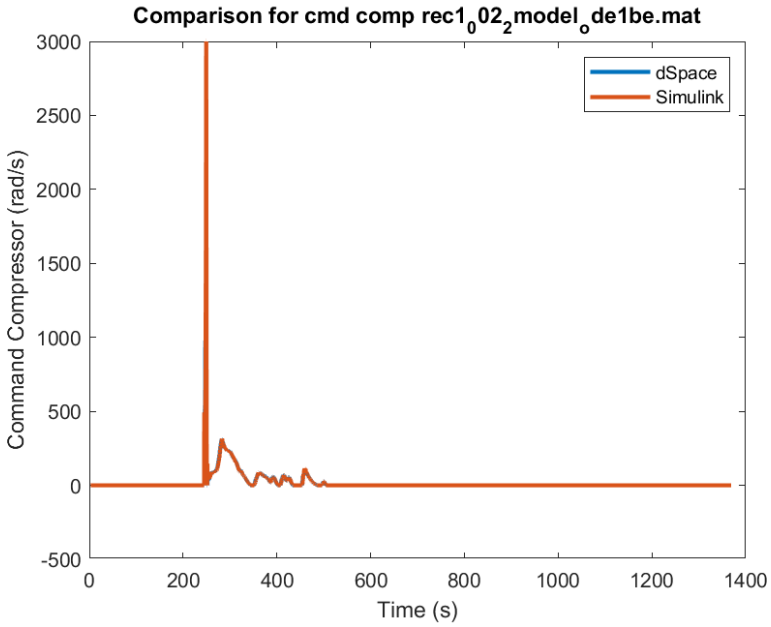


Figure 34: Double Model Simulation ode1be: Command Compressor

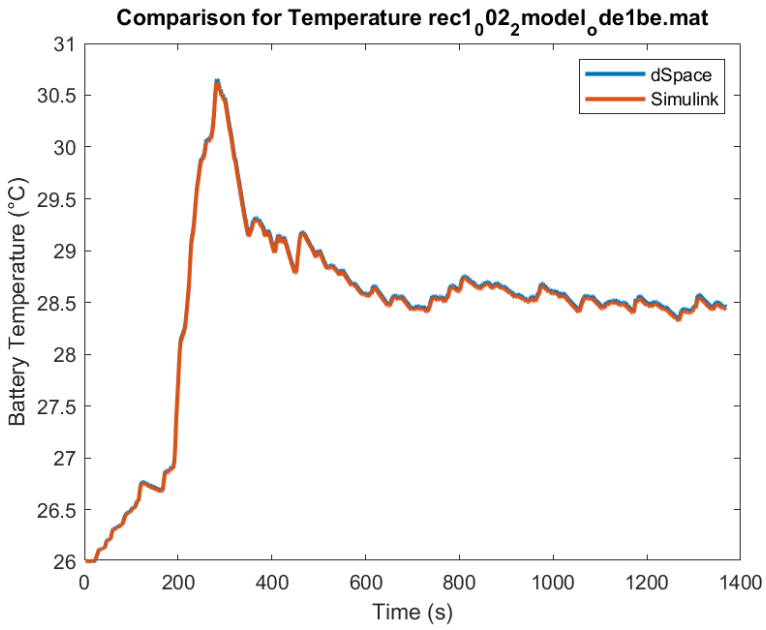


Figure 35: Double Model Simulation ode1be: Temperature

Parametro	EPower Compressor	Command Compressor	Temperature
RMSE	10.59 W	59.90 rad/s	0.020 °C
MAE	2.92 W	3.17 rad/s	0.018 °C
NMAE	1.28%	3.01%	0.06%
MAXAE	115.67 W	2324.01 rad/s	0.031 °C

Table 3: Results Double Model ode1be

For double model simulations, the results are better than the single model simulation, as highly expected, considering that a dedicated memory stack is used for each of the two full model parts (Controller and Plant). This makes the simulation much more efficient when the computational load is higher, like when the compressor is turned on.
Two different simulations were carried out because two different solvers have been analyzed for the Controller, the ode4 (Runge-Kutta 4) and the ode1be.
As mentioned before, an implicit solver has been chosen for all the simulations to deal with the stiffness of the Plant; anyway, whenever the Plant is decoupled from the Controller, other possibilities could be explored.
As a result, the differences between the two solvers are low and both can be used; moreover, since the whole system simulation was done with ode1be, a higher similarity with the offline simulation is justified.
For this setup all the errors (**RMSE, MAE, NMAE, MAPE**) diminished and, in some cases, were more than halved. (e.g. EPower Compressor (W) RMSE 25.82 against 10.59, MAE 6.39 against 2.92, considering Tables 1 and 3).
Temperature errors are a little higher but this can be due to little time misalignments that can be noticed when the error is that low (Figures 32,35).
As before, the bigger misalignments are present when the compressor is turned on, and the oscillations of the command are higher.
Anyway, these results show that splitting the full model into two different tasks significantly enhances real-time performance, ensuring more robustness and precision.

5 HiL Simulation with CAN Communication

After the first real-time simulations, a proper **HiL (Hardware in the Loop)** simulation is initialized. [22]
With Hardware in the Loop, a simulated Plant model, that, in this case, is uploaded to LabBox Scalexio, communicates with a Hardware dedicated to the Controller, which in this application is uploaded to MicroAutobox III.
Controllers, in real-time applications, after being simulated with their behavioral model, have to be tested within an ECU, so that residual mistakes in the Software or Hardware could be removed.

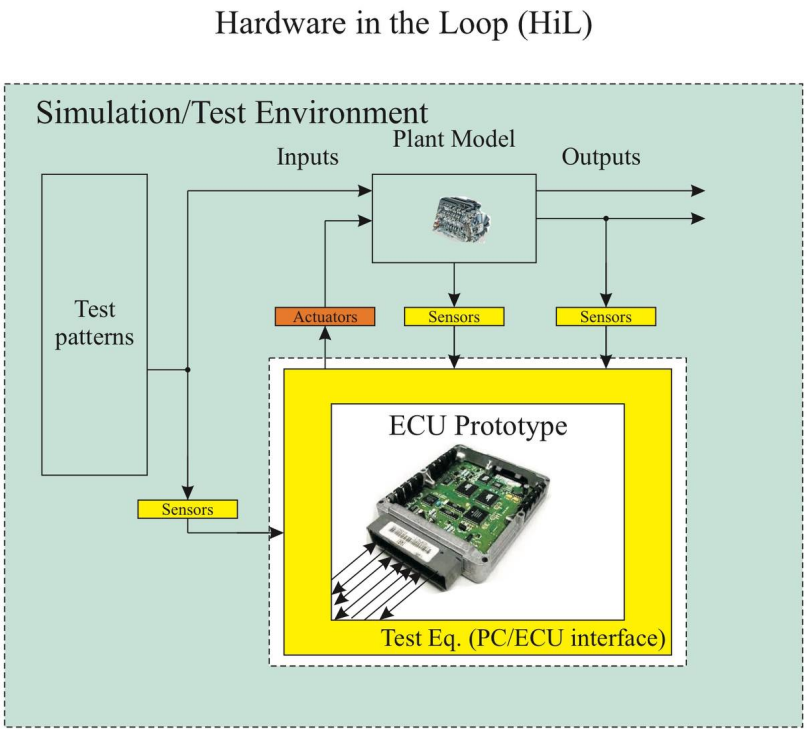


Figure 36: Hardware in the Loop (HiL) Scheme

Indeed, the main advantage is that with HiL simulations the proper Controller can be simulated without risking to damage the high-cost

Plant, having at the same time the possibility to test it in a wide range of conditions.
So, sensors acquisitions from the Plant go to the Controller that then, through actuators, controls the system.
To make this possible, a proper communication that has to be as realistic as possible and comparable with a real automotive environment has to be chosen.
CAN (Controller Area Network) is a communication technology that is widely used in the Automotive field. It is used alternatively to **LIN (Local Interconnect Network)** but CAN is faster and is used for more critical applications; actually CAN is used for ECU to ECU communication while ECU to actuator communication is done by LIN, which is cheaper and simpler.

In this application, the communication between the two dSPACE "ECUs" is implemented using CAN technology. As said before, this is much faster, and the LabBox Scalexio used in this work isn't equipped with a proper LIN board, but only with a CAN board.

5.1 Controller Area Network

CAN is a very reliable technology with an exchange rate that can go up to 1 Mb/s, excelling in terms of error recognition and management.[23]
It involves different layers:

- **Physical Layer:** it is the layer that transfers physically the bits on the shared medium (copper bus).
- **Medium Access Control (MAC):** It implements the protocols for the nodes to access and communicate within the medium.
- **Logical Link Control (LLC):** it is important in terms of the interface that creates the upper layers.

In terms of physical communication, bits are transmitted by a Non-Return to Zero (NRZ) coding scheme, which means that consecutive bits of the same logic value (High or Low) don't change level, so this makes this encoding method quite simple, but at the same time difficult to synchronize.
Synchronization is made possible every time the incoming signal has a transition from High to Low level.
Whenever the information coming through has 5 consecutive identical bits, Bit stuffing is used. After 5 consecutive identical bits, the opposite is inserted (high after 5 lows and vice versa). The system automatically drops the bit that comes after 5 identical bits and everything is synchronized.

CAN uses a Carrier Sense Multiple Access protocol, which means that there is no schedule, and whenever the bus is sensed free, the node can transmit. At the same time, it has an error detection and arbitration protocol, that, through a distributed algorithm, identifies the winner within the transmission by the presence of a certain dominant bit (Low voltage) rather than a recessive ones (High voltage).

Messages have different frames in it:

- **Data Frame**
- **Remote Frame**
- **Error Frame**
- **Overload Frame**

The **Data Frame** structure is made of a Start Of Frame (SOF), a single dominant bit that makes the bus status busy for all nodes. It is the bit that informs all the nodes that a communication is happening. Then there is the **Arbitration Field**: each frame has a different identifier with 11 bits in standard format and 29 bits in extended format. It is needed because when different nodes send different frames, the

one with the most consecutive dominant bits wins.
Remote Transmission Request indicates whether the frame carries Data (Dominant Frame) or is only a Remote Frame (Recessive Frame).
Then there is the **Control Field**: its main parts are IDE (Identifier Extension), which gives information on the CAN format (Standard or Extended) and Data Length Code, which gives information on the length of the Data frame.
Then there are the bits of the **Data Field** that can be up to 8 bytes.
Then there is the **CRC field** (Cyclic Redundancy Check) that is generated using a specific algorithm applied to the transmitted data. When the receiver calculates the CRC for the received data and finds it differs from the transmitted CRC, an error is detected.
In the end, there is the **ACK field** (Acknowledgment Field), which, with a single bit, tells whether a frame has been correctly received or not.

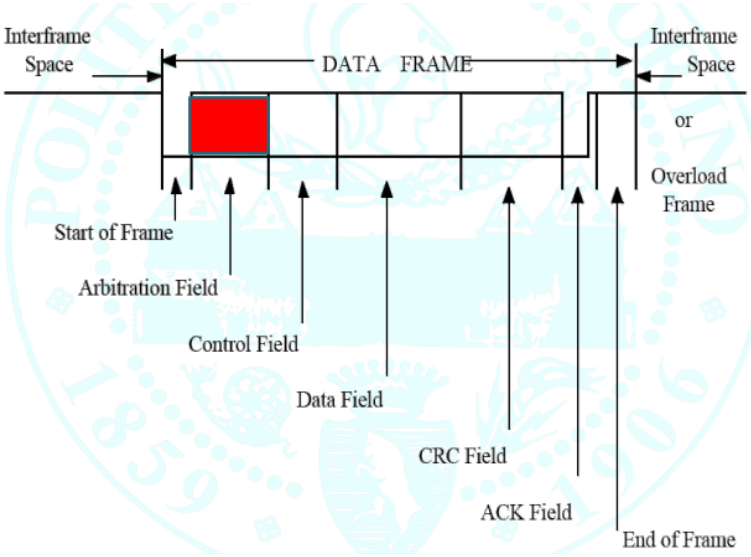


Figure 37: Data Frame

The **Remote Frame** is the same as the Data Frame but without the Data themselves, so, as said before, its role is data request, while the Data Frame is the response to the request.

The **Error Frame** is sent as a response to the Active Error Flag (6 consecutive dominant bits) generated by the bus every time an error is detected. The Error Frame is sent by the nodes when certain Bus errors arise:

- **Bit Error:** when a received bit is different from the one sent
- **Stuff Error:** after 5 identical bits the sixth isn't changed
- **Form Error:** a bit which has a certain expected value is wrong
- **CRC Error:** receiver CRC is different from the sender one
- **ACK Error:** no ACK delimiter in the ACK field

There is a sequence of specific bits that each node sends whenever an error is detected.

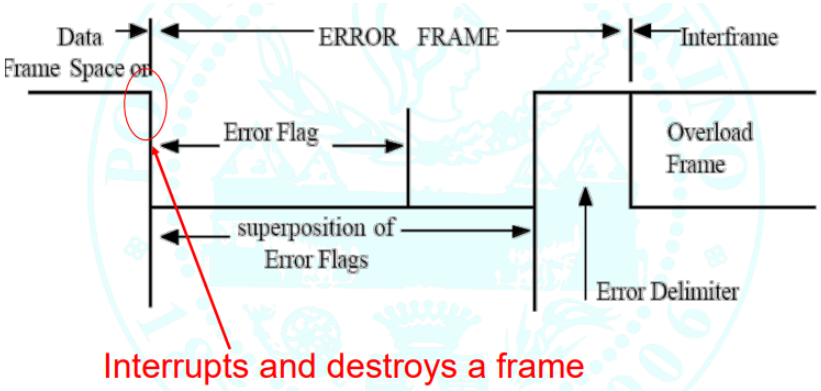


Figure 38: Error Frame

The **Overload Frame** is a sequence of 6 dominant bits that occupy the bus till the end of the Overload Frame (it comes after the Error Frame).

There is a counter that is updated every time an error is transmitted (**Transmit Error Counter - TEC**) or received (**Receive Error Counter - REC**), and there are different states the node can

be whether some thresholds are reached or not (as depicted in Figure 39). Each node can be in different states and, when an error is detected, the counter is increased, but when everything is fine, it is decreased.
In Error Active state, the condition is the basic one, in Error Passive state error flags can't be sent and, in Bus off, the node is disconnected and no transmission passes through it.

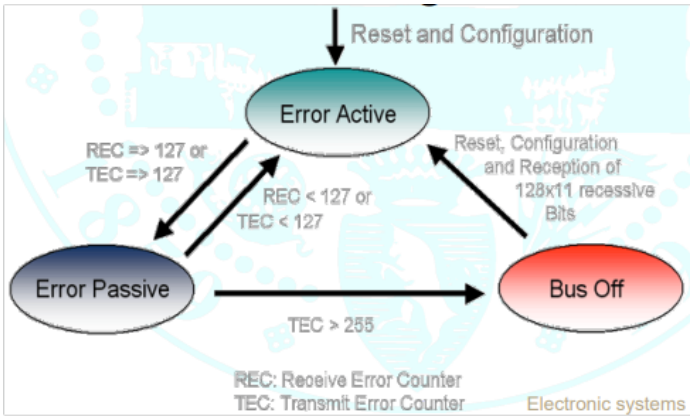


Figure 39: CAN Error Counters

5.2 Physical Requirements

A CAN bus requires physically 120 Ohm resistance between the CAN high and low pins, whether the pins are relative to the nodes that are placed at the bus ends. If the node is not at the end of the bus, this is not necessary; however, in this particular case, the nodes are two, the Plant (Scalexio) and the Controller (MicroAutobox III) and they are both at the end of the bus. So, as a first step, it is necessary to understand if this condition is met by the Hardware. By using dSPACE documentation, the physical characteristics of the CAN boards are displayed.

This is a channel of the CAN board 6341, the one installed in Scalexio:

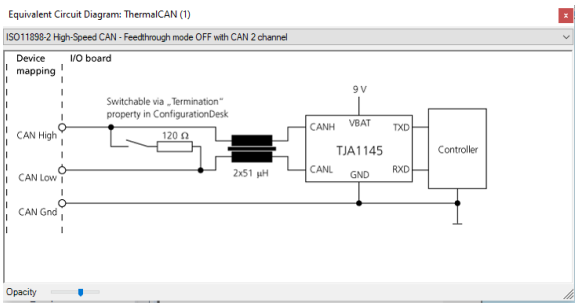


Figure 40: Scalexio CAN channel

While this is the configuration of the CAN channels of the DS1511, which is the board that is installed in the MicroAutobox III used in this study:

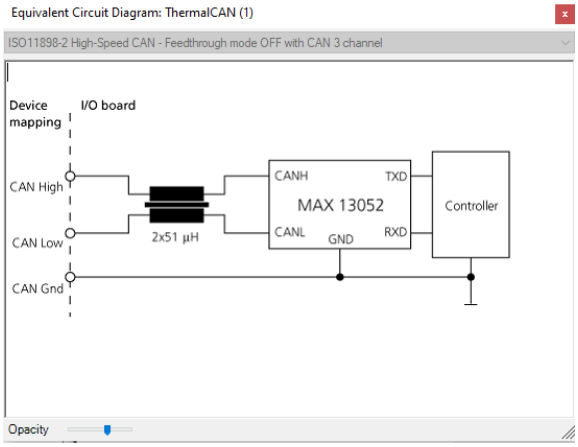


Figure 41: MicroAutobox III CAN channel

It can be observed that in Scalexio there is a termination that can be activated, as will be seen later, through ConfigurationDesk. This has been confirmed experimentally by manually measuring the resistance between the CAN high and low pins of both Hardware.

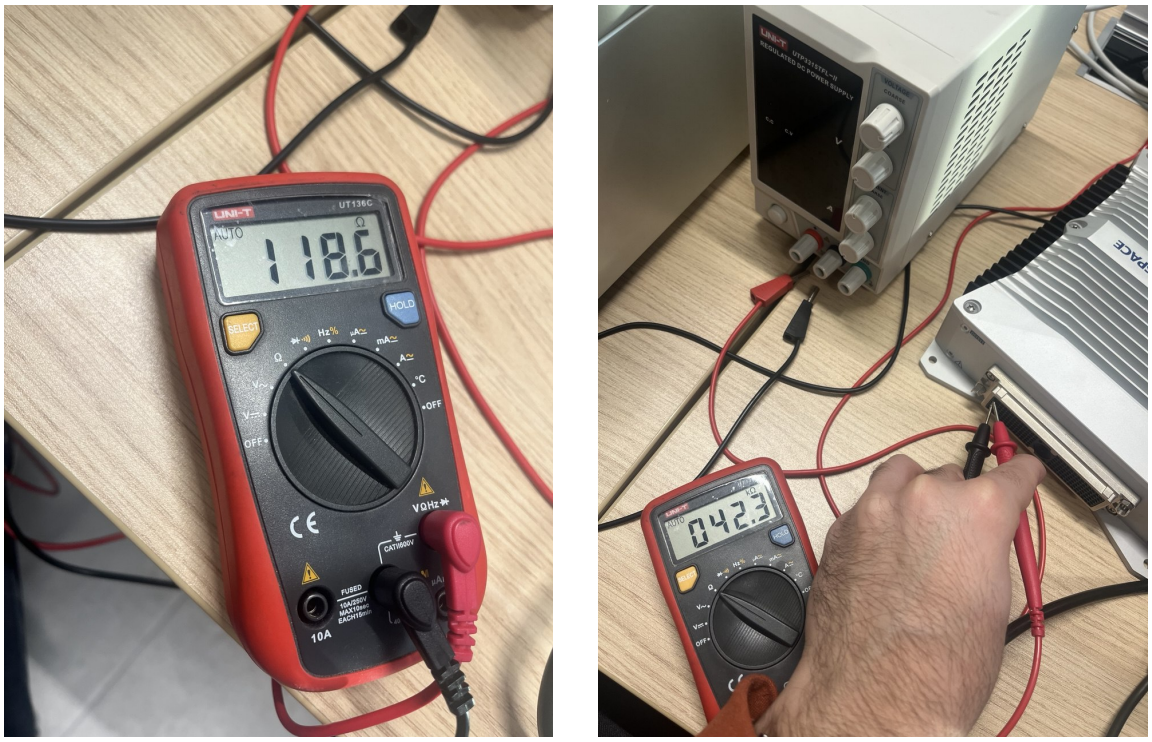


Figure 42: Multimeter: resistance measures

The first picture refers to the channels of the 6341 board and, as premised, it can be seen that the resistance is almost 120 Ohm, while in the other Hardware the resistance is higher than 42 kOhm because termination is missing.
So it was necessary to install a 120 Ohm resistance manually between the pins and this makes the communication, which would otherwise have been impossible, possible.
Resistances can even be welded by opening the Hardware in a proper slot, but in this case, an external resistance worked properly.

The CAN communication between Scalexio and MicroAutobox III has been implemented this way, particularly channel 1 of CAN board 6341 (Scalexio) was linked to channel 1 of the DS1511 board (MicroAutobox III).

5.3 Creation of a DBC file

Before creating a new project on dSPACE, it is necessary to configure the communication matrix of the CAN bus between the two Hardware. In this case, it is necessary to create a new DBC file. A **CAN DBC file** (CAN database) is a text file that contains information for decoding raw CAN bus data to "physical values". [24]
So it is essential to use CANdb++, a Software used to develop this kind of communication, with an interface that makes the process intuitive to people who are not used to this kind of Software.
Moreover, the tool is compliant with the automotive environment, which makes it perfectly suitable for this application.

To start with, when CANdb++ is opened, it is necessary to press "Create Database" and then select "CANTemplate.dbc"

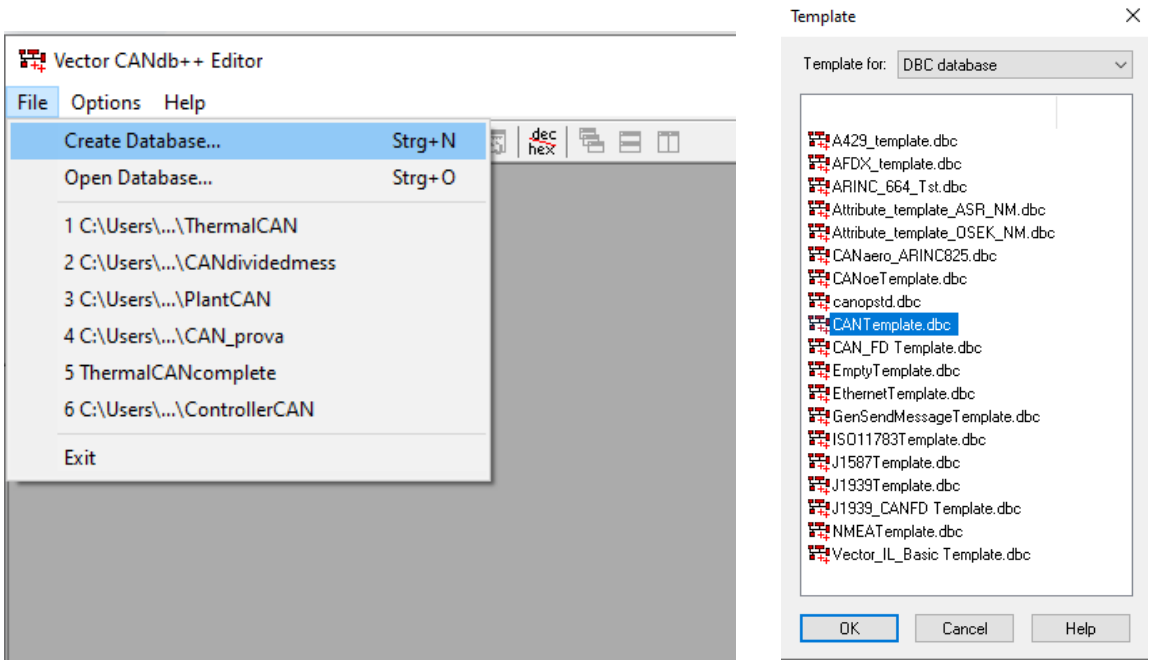


Figure 43: CANdb++ New Project

At first, signals should be configured. The signals that have to be generated are the ones that are present in the "Communication Net-

work" subsystem shown before in Simulink. Every signal has to be created, with specific parameters to be chosen accordingly.

By right-clicking on signals, a new one can be initialized like the one in the following example.

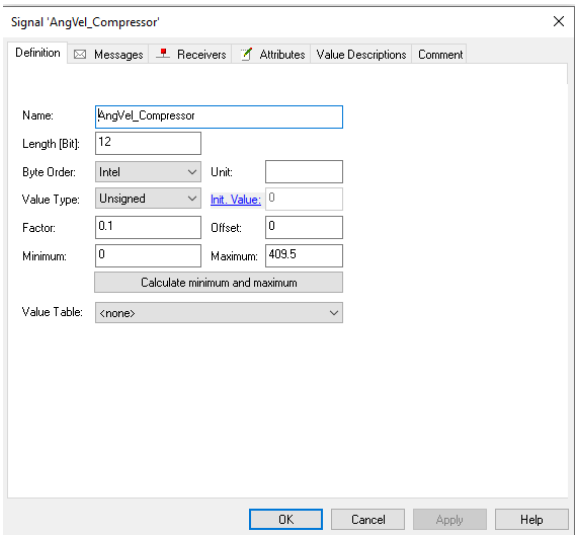


Figure 44: CAN Signal setting

The number of bits for a signal has to be chosen according to its range of values found in the simulations in Simulink, together with the factor that gives the resolution of the parameter. Then the type of signal is selected as in this case. Here "unsigned" was chosen because the values of the angular speed of the compressor are all positive. Accordingly, "signed" means that the values are both negative and positive (if there is an offset, that is the zero value for the signed signal).

After all the signals are created, it is necessary to create the messages, which are the ones that are sent through the CAN bus.
Messages have been created similarly to the signals.

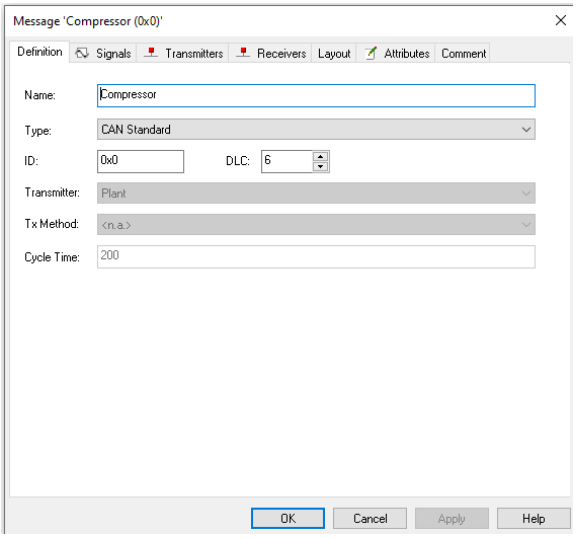


Figure 45: CAN Message settings

A message can carry a maximum Data of 8 bytes and has a unique address so that there is no mistake during the communication. Data length is expressed in the DLC field (Data Length Code). Then, in the message window, the signals can be added and arranged at will. By right-clicking on a specific message, in the "Edit Message" window, there is the "Layout" subwindow, that is used to change the order of the signals within the Data Field of the message.

As said before, the Data Field is 8 bytes (64-bit) long and this is the maximum length of signals that can be sent with a single message. The order and the starting bit of each signal can be arranged at will. This is how the layout of the message looks like:

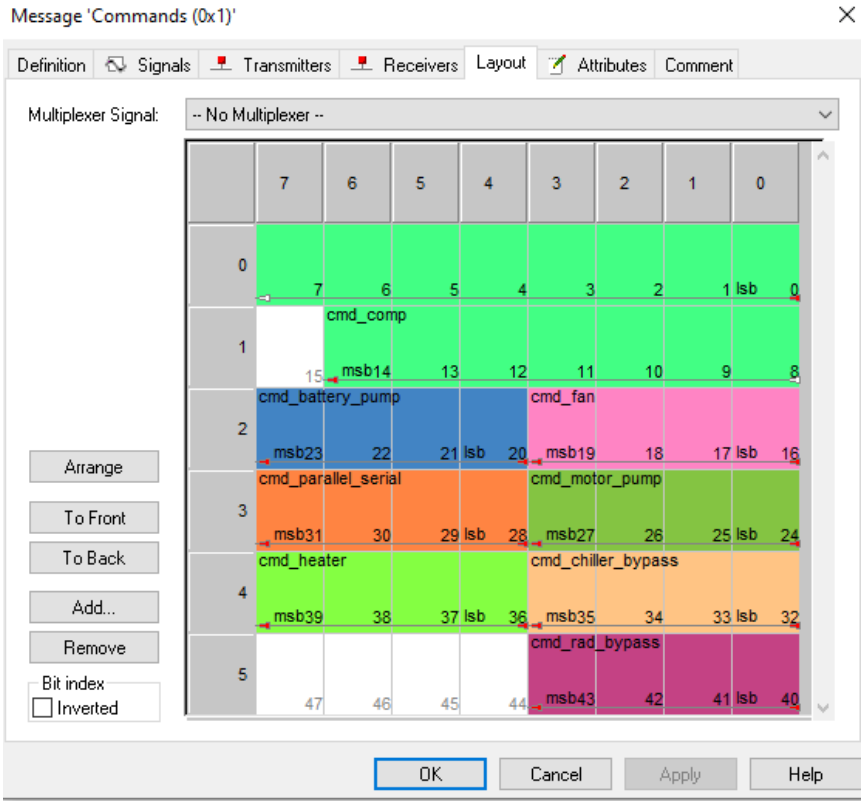


Figure 46: Message Layout Editor

Then, two nodes are initialized: the node relative to the Plant and the one relative to the Controller; in the window of the created node, the Tx (transmitted) and Rx (received) messages have to be assigned. By mapping the Tx messages for both nodes, the relative signals will be automatically assigned.

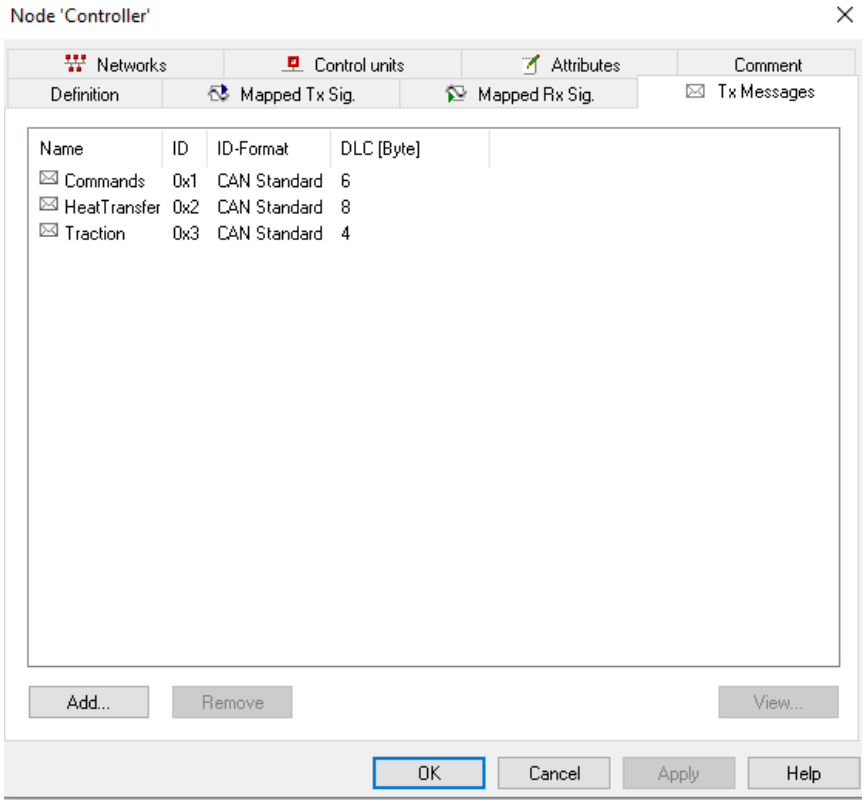
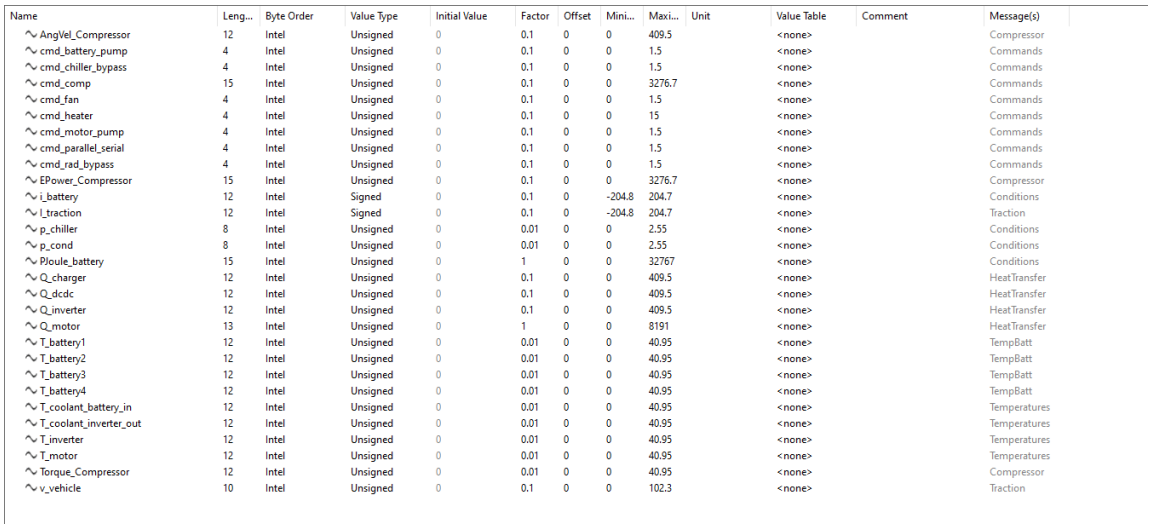


Figure 47: Controller: Tx Messages

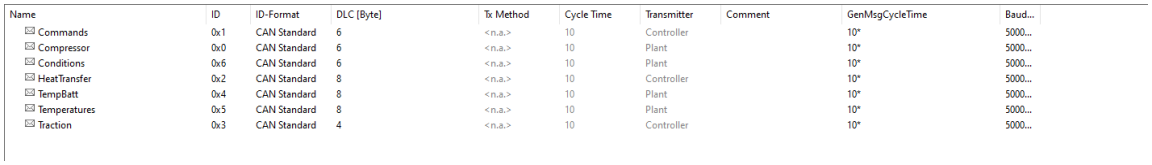
Then it is necessary to generate the attributes of the communication. This can be easily done by the window "View" and then by clicking on "Add Attributes". So the characteristics of the communication can be specified, like the Baud Rate (which represents the maximum amount of data that can be sent in 1 second) that has been set to 500 kbps and the GesMsgCycle Time (which defines the frequency of the messages). The latter can be modified at will on ConfigurationDesk.

Here is an overview of the signals and messages of the above-described DBC file:



Name	Leng...	Byte Order	Value Type	Initial Value	Factor	Offset	Mini...	Maxi...	Unit	Value Table	Comment	Message(s)
AngVel_Compressor	12	Intel	Unsigned	0	0.1	0	0	409.5		<none>		Compressor
cmd_battery_pump	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
cmd_chiller_bypass	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
cmd_comp	15	Intel	Unsigned	0	0.1	0	0	3276.7		<none>		Commands
cmd_fan	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
cmd_heater	4	Intel	Unsigned	0	0.1	0	0	15		<none>		Commands
cmd_motor_pump	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
cmd_parallel_serial	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
cmd_rad_bypass	4	Intel	Unsigned	0	0.1	0	0	1.5		<none>		Commands
EPower_Compressor	15	Intel	Unsigned	0	0.1	0	0	3276.7		<none>		Compressor
i_battery	12	Intel	Signed	0	0.1	0	-204.8	204.7		<none>		Conditions
I_traction	12	Intel	Signed	0	0.1	0	-204.8	204.7		<none>		Traction
p_chiller	8	Intel	Unsigned	0	0.01	0	0	2.55		<none>		Conditions
p_cond	8	Intel	Unsigned	0	0.01	0	0	2.55		<none>		Conditions
PJoule_battery	15	Intel	Unsigned	0	1	0	0	32767		<none>		Conditions
Q_charger	12	Intel	Unsigned	0	0.1	0	0	409.5		<none>		HeatTransfer
Q_dcdc	12	Intel	Unsigned	0	0.1	0	0	409.5		<none>		HeatTransfer
Q_inverter	12	Intel	Unsigned	0	0.1	0	0	409.5		<none>		HeatTransfer
Q_motor	13	Intel	Unsigned	0	1	0	0	8191		<none>		HeatTransfer
T_battery1	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		TempBatt
T_battery2	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		TempBatt
T_battery3	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		TempBatt
T_battery4	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		TempBatt
T_coolant_battery_in	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		Temperatures
T_coolant_inverter_out	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		Temperatures
T_inverter	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		Temperatures
T_motor	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		Temperatures
Torque_Compressor	12	Intel	Unsigned	0	0.01	0	0	40.95		<none>		Compressor
v_vehicle	10	Intel	Unsigned	0	0.1	0	0	102.3		<none>		Traction

Figure 48: Signal overview: DBC file



Name	ID	ID-Format	DLC [Byte]	Tx Method	Cycle Time	Transmitter	Comment	GenMsgCycleTime	Baud...
Commands	0x1	CAN Standard	6	<n.a.>	10	Controller		10*	5000...
Compressor	0x0	CAN Standard	6	<n.a.>	10	Plant		10*	5000...
Conditions	0x6	CAN Standard	6	<n.a.>	10	Plant		10*	5000...
HeatTransfer	0x2	CAN Standard	8	<n.a.>	10	Controller		10*	5000...
TempBatt	0x4	CAN Standard	8	<n.a.>	10	Plant		10*	5000...
Temperatures	0x5	CAN Standard	8	<n.a.>	10	Plant		10*	5000...
Traction	0x3	CAN Standard	4	<n.a.>	10	Controller		10*	5000...

Figure 49: Messages overview DBC file

GenMsgCycleTime was originally set to 10 Hz (as it can be seen), but then, in ConfigurationDesk, 5 Hz (0.2s) were used.

5.4 dSPACE initialization

After the CAN communication matrix is configured and the DBC file is ready to be used, a new project within ConfigurationDesk can be started.
Before creating a new project, both Hardware must be connected by the Ethernet cable to the PC, so that both can be online at the same time.

This time a different initialization has to be made: it is necessary to use two different applications when starting ConfigurationDesk. So at first, one of the two models (Plant or Controller) is uploaded in ConfigurationDesk with its proper Hardware (Scalexio LabBox and MicroAutobox III respectively), then the same operation is repeated for the other application. Theoretically, it is possible to create multiple Hardware applications, but only with the proper license, which in this case was missing.

When ConfigurationDesk is started, it is necessary to go to the "Buses" interface to configure the CAN communication.
While in the "Buses" window, in the "Buses" subwindow (Bottom left part of Figure 50), it is necessary to press "Add Communication Matrix" by right-clicking and then, after adding the DBC file, drag and drop the communication file on the "Bus Configurations" section.

Then, it is necessary to go to "Bus Access Requests" and, at first, by right-clicking on "Bus Access Request" (which can be seen under ThermalCAN in Figure 51) "Automatic Bus Access Assignment" should be selected. With this, a CAN virtual channel is generated (ThermalCAN(1)): by right-clicking again on this virtual channel, the actual channel (of the Hardware) that has been physically used for CAN communication must be selected (channel 1 in this case for both Hardware).

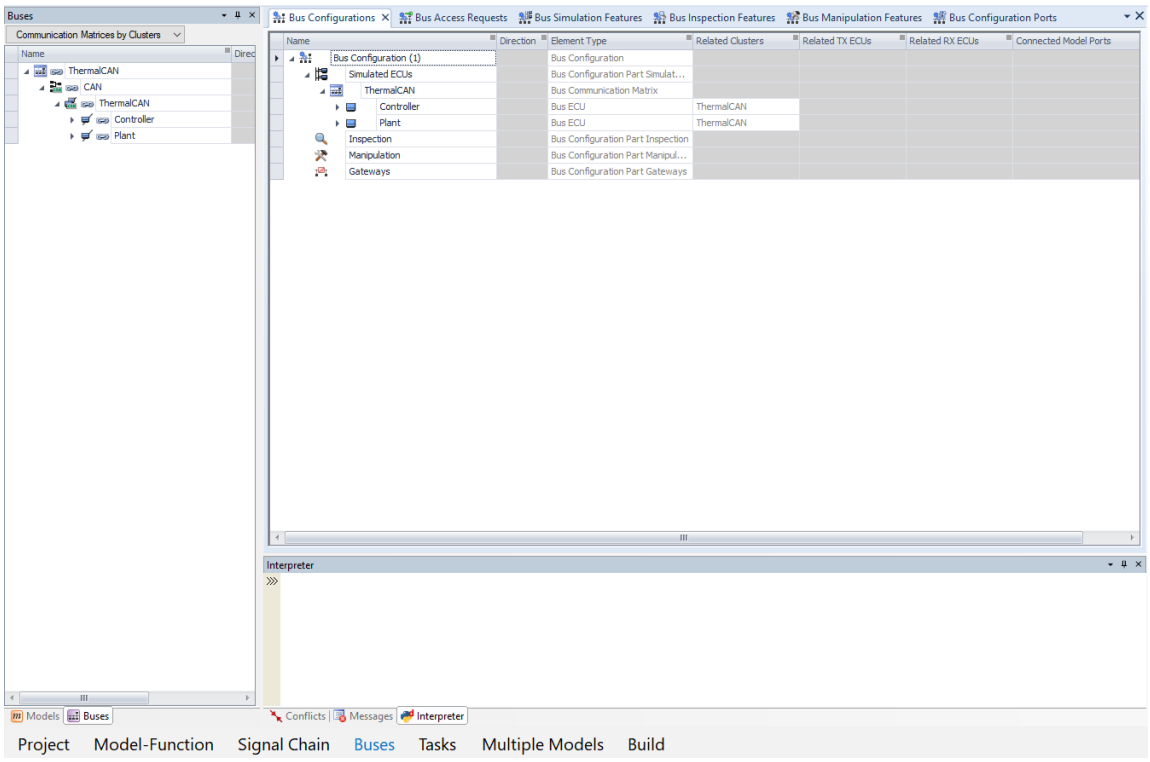


Figure 50: ConfigurationDesk: Bus Interface

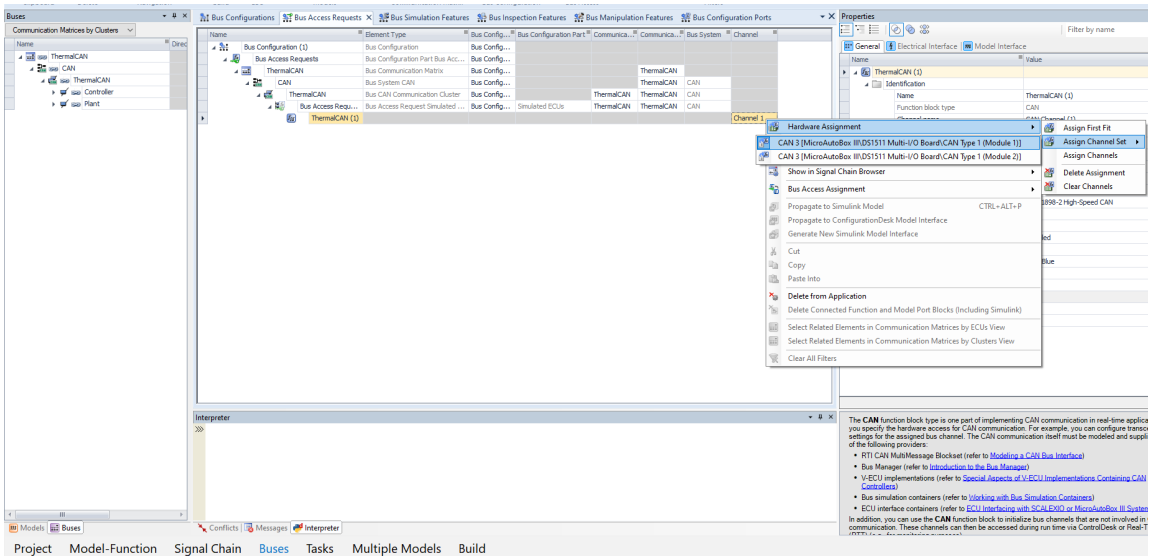


Figure 51: ConfigurationDesk: Bus Access Request

After that, it is necessary to go to "Bus Configuration Ports" and link the homonymous signals (through drag and drop) to the related blank "Connected Model Ports" space. Signals should be connected only to the parts related to the ECU (Plant or Controller) of the actual application. Then, these passages should be repeated for the other ECU. After that, the "Test Automation support" should be "Enabled" for all the ports. This will make possible the analysis of the communication status between the nodes on ControlDesk.

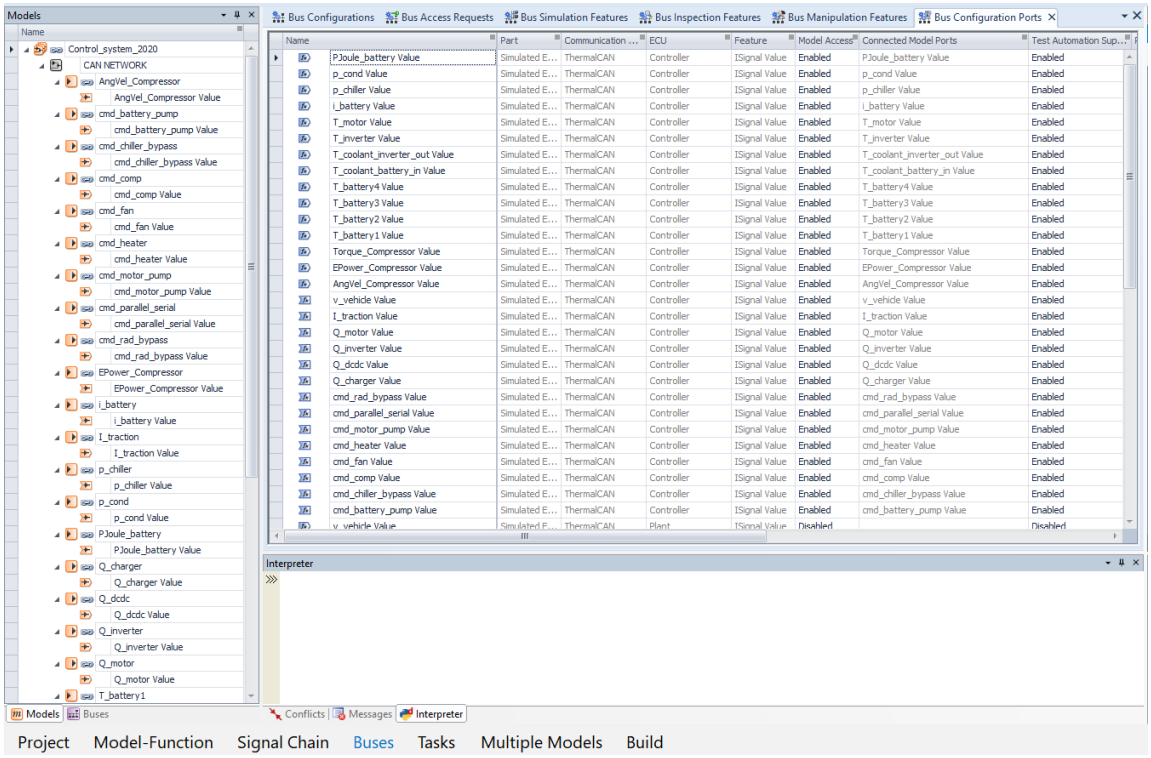


Figure 52: ConfigurationDesk: Bus Configuration Ports

This is what the model looks like in the "Signal Chain" window (Figure 53). In the "Functions" subwindow there is "Bus Configuration", a block that represents what has been described up to now. So all the arrows that link this block with the model's interface represent what has been done in "Bus Configuration Ports".
"ThermalCAN" function simply represents the physical CAN channel; CAN High, Low, and GND are not linked because they are natively

connected with Hardware since the one used is dSPACE Hardware. Anyway, this is an interesting function because it makes us see that this kind of communication can be interfaced even between dSPACE Hardware and an external Hardware.

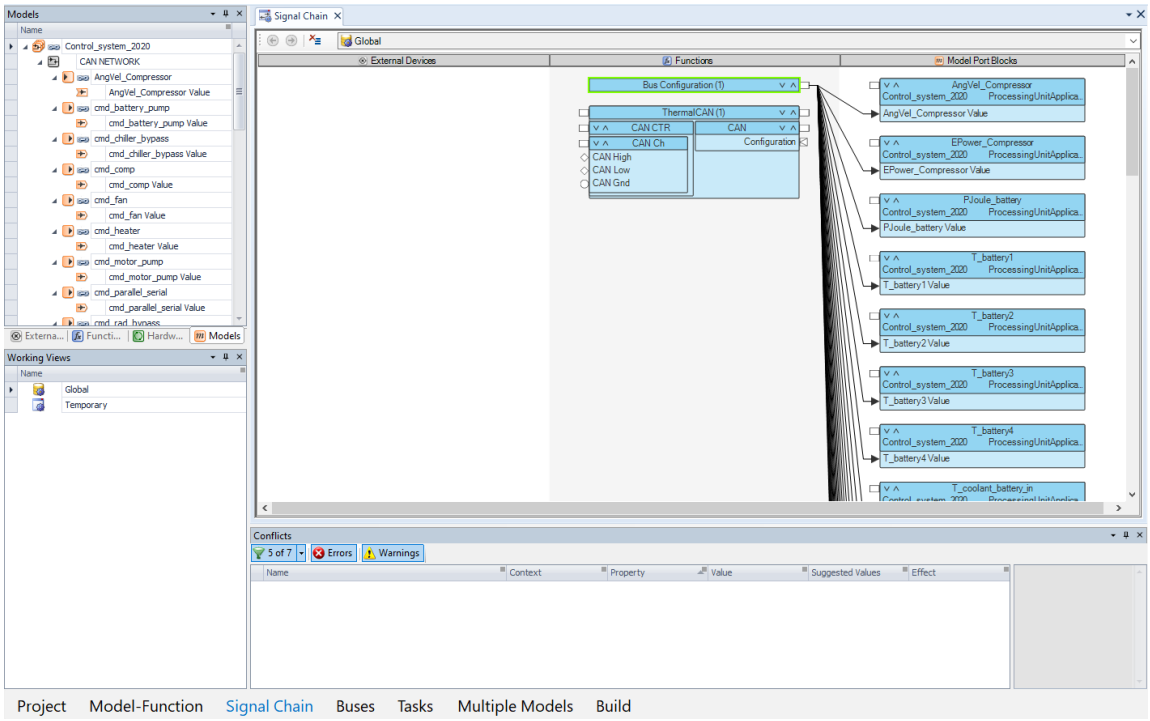


Figure 53: ConfigurationDesk: Signal Chain, CAN project

Afterwards, by right-clicking on ThermalCAN, termination has to be set at "ON" for the Scalexio LabBox application, while it has to be set at "OFF" for the MicroAutobox III application, considering that, as mentioned before, there is not an internal termination but the resistance was added externally. Other parameters can be adjusted in this window, like Baudrate, which was defined before.

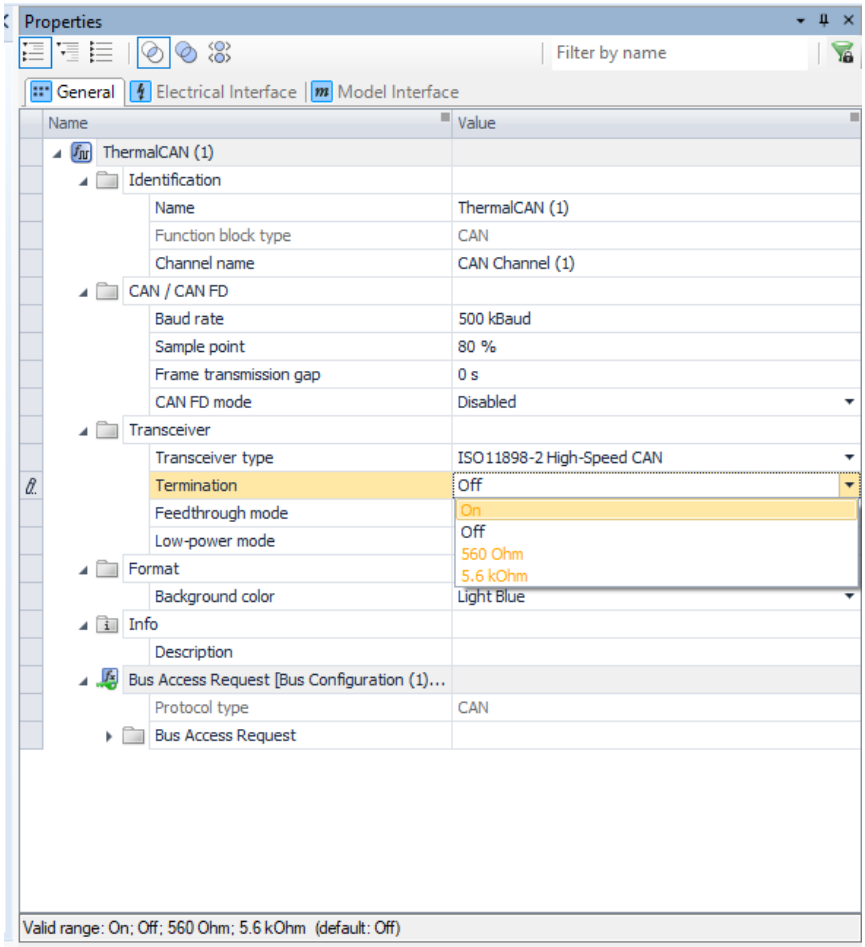


Figure 54: Turning ON/OFF physical terminations

So everything is ready for the Build of the new configuration, both for the Controller and the Plant. Here the reasoning is quite similar to what has been explained before but with some differences.

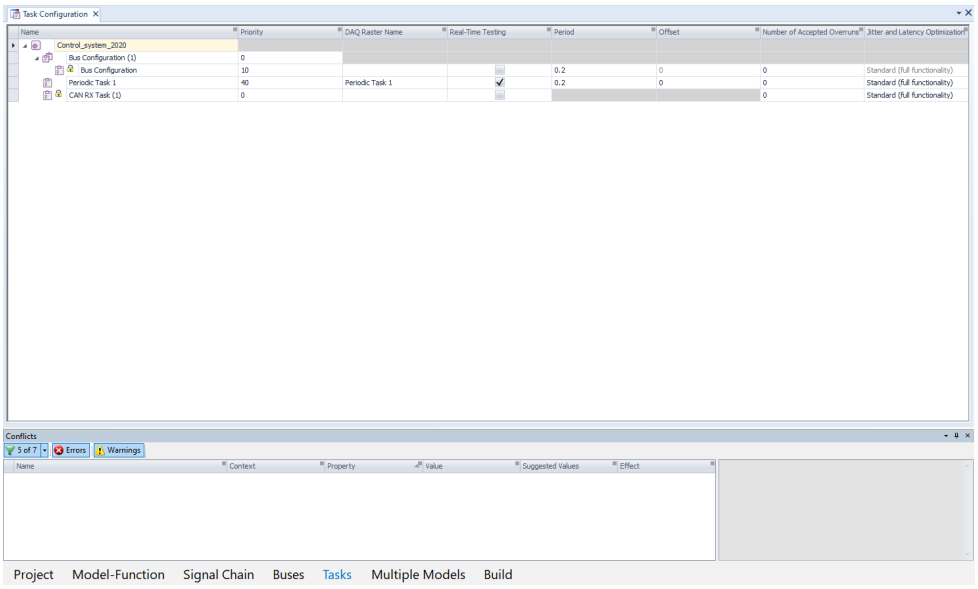


Figure 55: ConfigurationDesk: Tasks, CAN project

It can be seen in Figure 55 that there are more than two tasks for each application and, in this case, a higher priority was considered for the Periodic Task (Plant or Controller model) because is the one that has the higher computational burden, so, to avoid miscalculations, it has to be that high; a little priority has to be configured for Bus Configuration: if put to 0 there would be no communication and so no signal would flow within the communication network. CAN RX task is created by the Software to configure the transfer and it isn't modified. According to dSPACE documentation "CAN RX Task is a type of task which calculates to do filtering by Software on the processor whether to transfer a CAN event to the application or not based on the received data".

After the build is done and the two applications are loaded on the Hardware, a new Experiment can be opened within ControlDesk.

In ControlDesk, the operations are the same but, after the first Experiment + Application is opened, it is necessary to right-click on "Hardware Configurations" and select "add Platform/Device" to add

within the same Experiment the application linked to the second device.

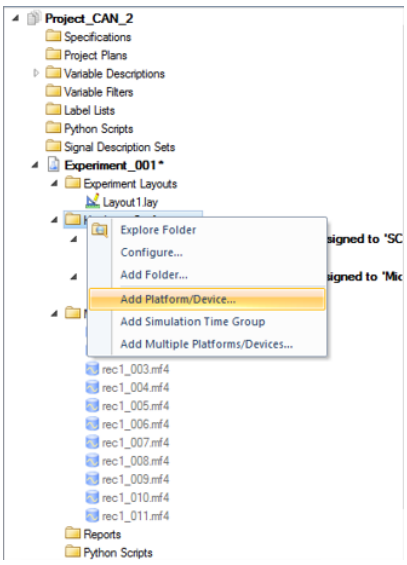


Figure 56: ControlDesk: Double Device Experiment

Then everything is set up and the application, after a proper layout is defined, can start as before seen.

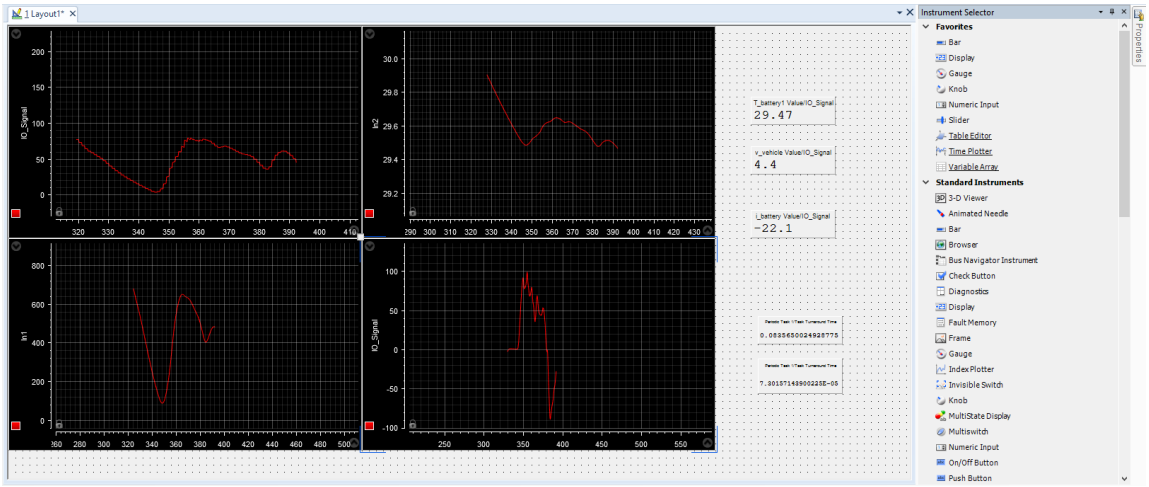


Figure 57: ControlDesk: Operational Interface

It is possible to add another device: CAN device. Indeed, in devices, it is possible to click on "CAN Bus Monitoring" and then, in the

selection of variable description, choose "CAN generator" that can be found in "Demos" among dSPACE documents.

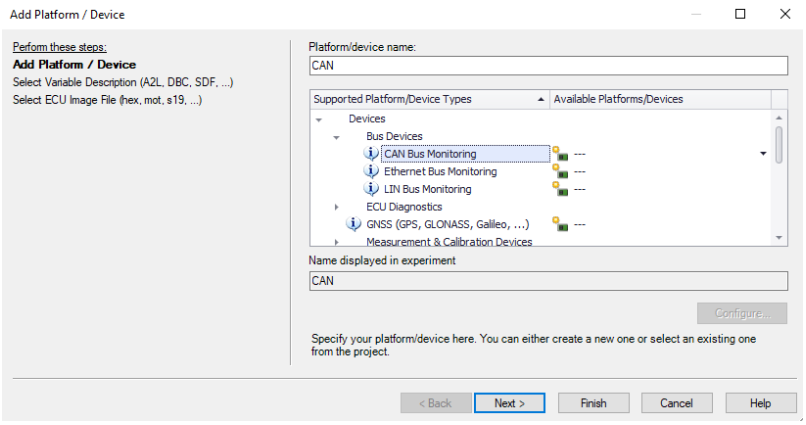


Figure 58: ControlDesk: CAN device

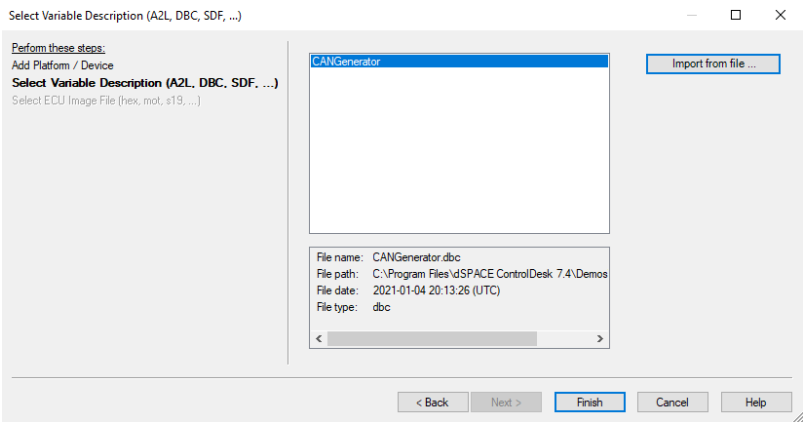


Figure 59: ControlDesk: CAN generator

Then, with all the devices set up, it is possible to go to the sub-window "Bus Navigator" and, by right-clicking on "CAN Controller", "Add Monitor" should be pressed: this way a new layout window is opened and all the messages that come and go through the CAN bus can be analyzed together with the payload. After the discussion of the results, the CAN payload for these operations will be presented.

5.4.1 Results

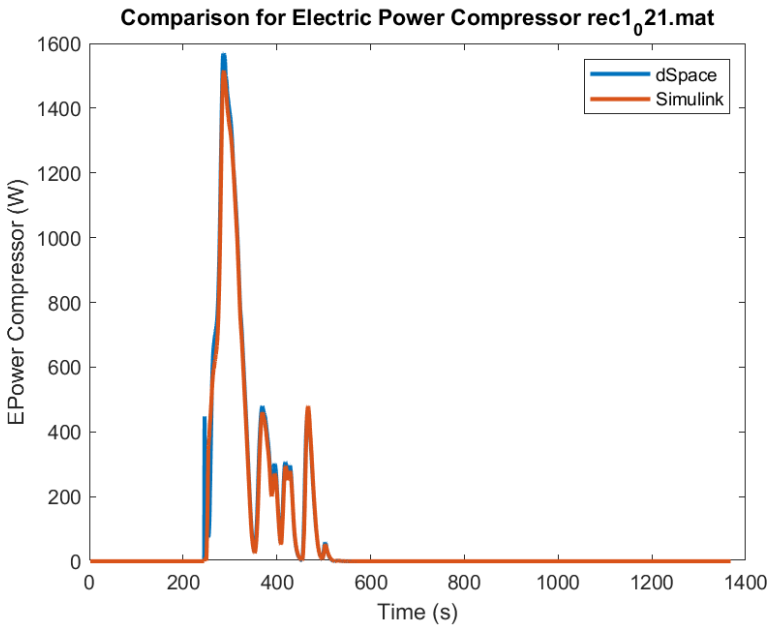


Figure 60: HiL simulation: EPower Compressor

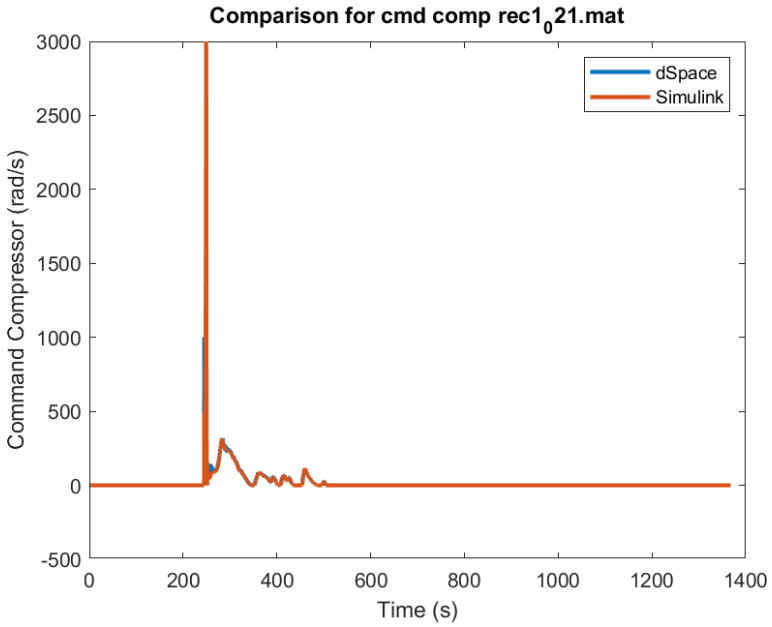


Figure 61: HiL simulation: Command Compressor

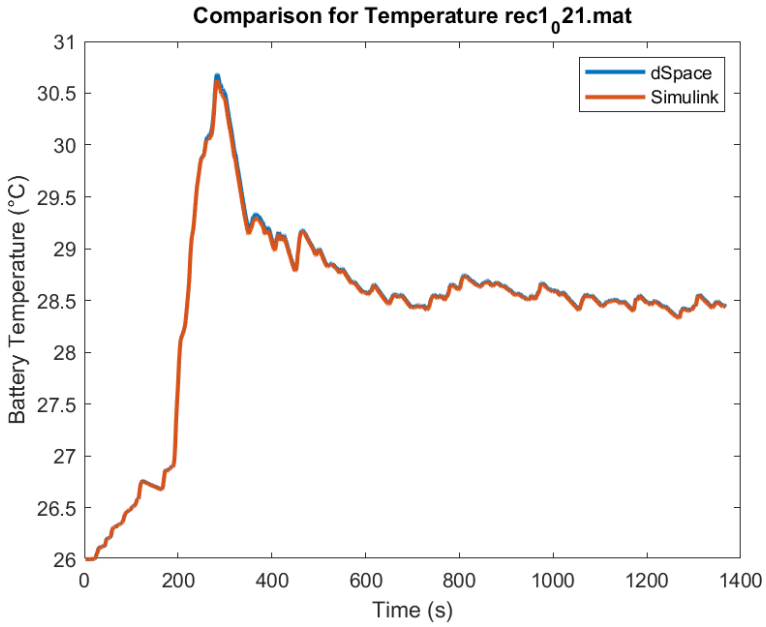


Figure 62: HiL simulation: Temperature

Parametro	EPower Compressor	Command Compressor	Temperature
RMSE	25.74 W	87.91 rad/s	0.018 °C
MAE	6.06 W	5.39 rad/s	0.013 °C
NMAE	2.66%	5.13%	0.05%
MAXAE	342.85 W	2968.06 rad/s	0.065 °C

Table 4: Results Hardware in the Loop

As far as the results for the whole simulation system with CAN communication are involved, it can be noted that they are very similar in terms of errors to the first simulation with the single model.
The graphs are quite similar and the behavior is the same, when the compressor is turned on, an initial peak of the Electric Power that isn't obtained in the Simulink results, can be observed in the dSPACE simulation (Figure 60).
While results are quite similar to the single model, the reasons for the differences have to be found in the latencies that a CAN bus can introduce in the communication between Plant and Controller.

Anyway, results are acceptable, and the ability of the Controller to match the **Target Temperature** is unchanged (Figure 62).
The fact that Temperature is, for every simulation, almost errorless, gives another hint of the fact that slow dynamics, such as those of thermal systems, are more faithfully reproduced and less affected by overruns and latencies than more dynamically changing parameters like the ones related to the compressor.
Skipping some operations is intuitively much less detrimental for slow dynamics such as thermal ones, because the system evolves over longer time scales.

5.5 Analysis of CAN payload and Controller Task TurnAround Time

After the analysis of the parameters, it is convenient to analyze the CAN payload and the messages that are being exchanged while the application is online. As mentioned before, within the ControlDesk application, it is possible to analyze the bus load:

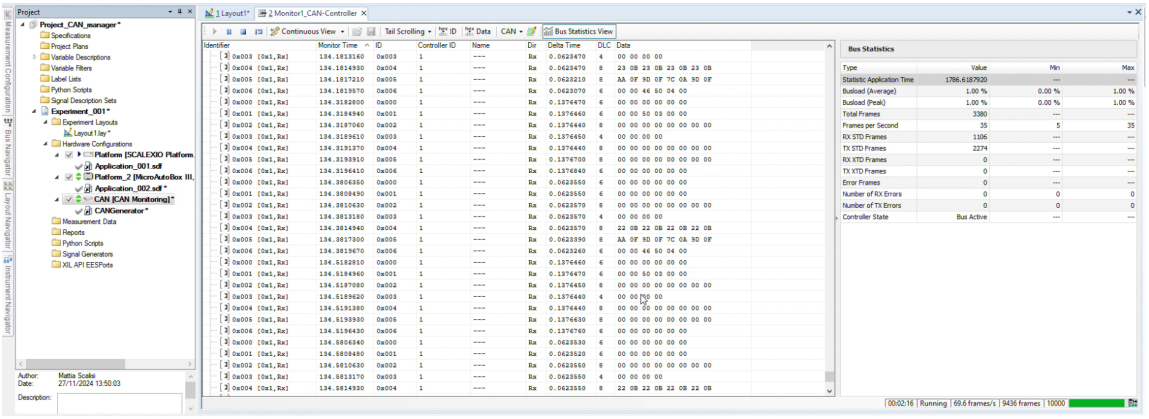


Figure 63: Analysis of the CAN payload and messages

It can be seen that there are 35 frames exchanged per second, which is what was expected: indeed, there are 7 messages exchanged only 5 times per second.
It can be seen that both the **Peak** and **Average Bus Load** are 1.0%.

Considering that these messages are CAN standard messages, which include an Overhead of 61 bits and an average Data field of 40-45 bits for each message, an average weight of 105 bits can be assumed. Considering the formula:

$$\text{Bus Load} = \frac{\text{Bit per cycle} \times \text{Message Frequency} \times 100}{\text{CAN BaudRate}} \tag{36}$$

$$\text{Bus Load} = \frac{7 \times 105 \times 5 \times 100}{500000} = 0,735\% \tag{37}$$

The result turns out to be compatible with what could be theoretically expected, and what has been obtained is much lower than the maximum payload. This is an advantage for possible further improvements of the model, even with an increased frequency or an increased number of exchanged messages.
Anyway, with this controller it would not be necessary; on the contrary, this number could be further reduced considering the timestep of the MPC.

Speaking of the actual Controller, that is the main interest of this work, the actual task period is already far below the control sample time of 1 second: the simulation was indeed carried out for 0.2 seconds. That considered, the **Task Turnaround Time** related is much lower than the timestep, even with an implicit solver like ode1be as can be seen in the picture below.

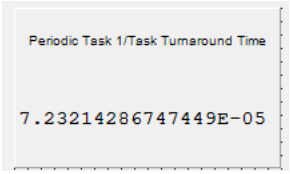


Figure 64: Controller Task Turnaround Time

6 Thermal Model with Custom Components

6.1 Simscape Components: Compressor and Fan

For the final simulations new components in Simscape language are used in place of the old compressor and fan.
It was necessary to create new custom Simscape components to have an interface that provides the variables required for the control system more directly and, at the same time, to improve their functionality on the way of the components used in the newer Matlab versions.
Simscape components have some typical sections within their code that characterize them in all their peculiar functions:[25]

- **Nodes:** These are the links of the component with the rest of the system, they are specifically part of a certain physical domain and they have to be defined in function of it. The picture below refers to the nodes of the new fan that operates in the mechanical and moist air domain.

```
nodes
    A = foundation.moist_air.moist_air; % A:left
    B = foundation.moist_air.moist_air; % B:right
    R = foundation.mechanical.rotational.rotational; % R:left
    C = foundation.mechanical.rotational.rotational; % C:right
end
```

Figure 65: Simscape: Nodes

- **Parameters:** These are the parameters that can be modified through the Simulink interface. Parameters can then be ordered in different sub-interfaces and can be made private with a proper annotation (parameters set as non-modifiable by the user). Here there are some parameters that have been used in the new compressor component.

```
parameters
    displacement = {80,    'cm^3/rev'}; % Displacement volume
    isentropic_efficiency = {0.65,    '1'}; % Isoentropic efficiency
    mechanical_efficiency = {0.9,    '1'}; % Mechanical efficiency
    area_A    = {0.01, 'm^2' }; % Inlet area at port A
    area_B    = {0.01, 'm^2' }; % Outlet area at port B
end
```

Figure 66: Simscape: Parameters

- **Variables:** These are the variables that are here defined in terms of initial conditions and measure unit and that are then updated for each timestep according to the equations after defined. As state variables, their values are constantly updated and can be monitored externally. Here in the example, there are some variables initialized for the compressor.

```
variables (Access = protected)
    mdot_A = {0, 'kg/s'}; % Mass flow rate into port A
    mdot_B = {0, 'kg/s'}; % Mass flow rate into port B
    Phi_A  = {0, 'kW'  }; % Energy flow rate into port A
    Phi_B  = {0, 'kW'  }; % Energy flow rate into port B
end
```

Figure 67: Simscape: Variables

- **Intermediates:** These are variables that are defined but only used for internal calculations, so they're not accessible from the outside and they're not constantly saved. They're mainly used for intermediate calculations to find the main variables. Here's an example.

```
intermediates (Access = private, ExternalAccess = none)
    % Compute isoentropic enthalpy increase and change in specific entropy
    % Valid for flow rate A -> B
    [Ds, ht_in, ht_out_is] = foundation.two_phase_fluid.sources.isentropic_relation( ...
        A.p, B.p, u_in, u_out, mdot_A, area_A, area_B, ...
        A.u_min, A.u_max, A.unorm_TLU, A.p_TLU, A.v_TLU, A.s_TLU, A.u_sat_liq_TLU, A.u_sat_vap_TLU);
end
```

Figure 68: Simscape: Intermediates

- **Branches:** These define how the variables flow between the component ports, so by them, there is an association between the variables and the physical characteristics of the port as defined in nodes. For example, if I have a "Moist air port", a variable linked with the flow rate of the fluid is expected to be defined (e.g.mdot_A) and then linked here to the physical flow rate (e.g. A.mdot).

```
branches
    mdot_A : A.mdot -> *;
    mdot_B : B.mdot -> *;
    Phi_A  : A.Phi  -> *;
    Phi_B  : B.Phi  -> *;
end
```

Figure 69: Simscape: Branches

- **Equations:** These regulate the physical behavior of the system, linking variables with each other through equations that define important physical laws (like conservation of mass and energy). Moreover, they are used to define the physical constraints of the system: minimum and maximum values (temperature, pressure, efficiency) to ensure an always realistic behavior of the component and the feasibility of its interactions with the rest of the system. Here is an example

```
equations
    % Mass balance
    mdot_A + mdot_B == 0;

    % Energy balance
    Phi_A + Phi_B + fluid_power == 0;

    %Mechanical Power
    P_mech == fluid_power / mechanical_efficiency;

    %Mass Flow Rate output
    m_dot == mdot_A

    % Run-time variable checks
    assert(A.p >= A.p_min, message('physmod:simscape:library:two_phase_fluid:PressureMinValid', 'A'));
    assert(A.p <= A.p_max, message('physmod:simscape:library:two_phase_fluid:PressureMaxValid', 'A'));
    assert(A.u >= A.u_min, message('physmod:simscape:library:two_phase_fluid:InternalEnergyMinValid', 'A'));
    assert(A.u <= A.u_max, message('physmod:simscape:library:two_phase_fluid:InternalEnergyMaxValid', 'A'));
    assert(B.p >= B.p_min, message('physmod:simscape:library:two_phase_fluid:PressureMinValid', 'B'));
    assert(B.p <= B.p_max, message('physmod:simscape:library:two_phase_fluid:PressureMaxValid', 'B'));
    assert(B.u >= B.u_min, message('physmod:simscape:library:two_phase_fluid:InternalEnergyMinValid', 'B'));
    assert(B.u <= B.u_max, message('physmod:simscape:library:two_phase_fluid:InternalEnergyMaxValid', 'B'));
end
```

Figure 70: Simscape: Equations

- **Annotations:** These are graphical notes about, for example, the placement of the port in Simulink or the arrangement of the parameters in the interface.

```
annotations
    A : Side = left;
    B : Side = right;
    N : Side = bottom;
    Icon = 'my_compressor.svg';
end
```

Figure 71: Simscape: Annotations

6.1.1 New Compressor

This new compressor model represents a volumetric compressor that manages, by a speed input, the flow rate of the fluid from A to B: this makes it similar, logically, to a controlled mass flow rate.
It presents, as ports, two refrigerant ports (A and B), the mass flow rate, and the mechanical power that makes the calculation of the torque quite straightforward.
Some essential equations can be highlighted to understand the in-depth behavior of the compressor.

The **mass flow rate** is controlled, as said, as a function of the speed input which is the manipulated variable of the control logic, and of the volumetric efficiency; its equation is the following:

$$\dot{m}_A = \eta_v \cdot V_s \cdot \frac{N}{v_{in}} \tag{38}$$

where $\dot{m}_a$ is the mass flow rate through port A, η_v is the volumetric efficiency, V_s is the displacement volume, N is the input speed expressed in rpm and the latter v_{in} is the specific volume of the entering fluid.

Then, there is the **energy equation** that accounts for the internal enthalpy increase of the fluid, considering that the compressor by

definition adds energy to it. This equation considers the increase in enthalpy due to the compression, and the one due to the losses during the process.

$$h_{\text{out}} = h_{\text{in}} + \frac{h_{\text{out,is}} - h_{\text{in}}}{\eta_{\text{is}}} \tag{39}$$

This equation is needed because the compressor isn't isentropic and so it is necessary to correct the ideal enthalpy to account for this non-ideality of a real compressor.

The output **mechanical power** is obtained through mechanical efficiency as before described:

$$P_{mech} = \frac{\dot{m}_A(h_{\text{out}} - h_{\text{in}})}{\eta_{\text{mech}}} \tag{40}$$

Then there are common equations of **mass and energy conservation** between the two compressor ports:

$$\dot{m_A} + \dot{m_B} = 0 \tag{41}$$

$$\phi_A + \phi_B + \dot{m}_A(h_{out} - h_{in}) = 0 \tag{42}$$

Then there is an equation for the **volumetric efficiency** that takes into account the fact that this efficiency isn't constant and varies as a function of the difference between the nominal volume ratio, which is a quantity that expresses how much the fluid is compressed under nominal conditions, and the actual volume ratio, which takes into account the operating conditions:

$$\eta_v = 1 + C - C \cdot \frac{v_{in}}{v_{out}} \tag{43}$$

Where:

$$C = \frac{1 - \eta_{v,nom}}{\frac{v_{nom,in}}{v_{nom,out}} - 1} \tag{44}$$

6.1.2 New Fan

This fan was modeled to follow the behavior of the fan component implemented in the newest Simscape version.
It presents 2 moist air ports (A and B) and 2 mechanical rotational conserving ports (R and C).
In this case, a properly set fan was essential because the one that was modeled after Mathworks didn't work properly. The fan principally creates a pressure difference that, creating a continuous airflow, cools more efficiently the system.

The **pressure drop characteristic curve** of this fan has been modeled by using a quadratic equation.

$$dp_{ch} = a \cdot Q^2 + b \cdot Q + c \tag{45}$$

Where Q is the volumetric flow rate, dp is the pressure drop, and a,b,c are coefficients calculated as a function of the reference and zero flow conditions of Q and dp.

Similarly, the **fan efficiency characteristic curve** is modeled:

$$\eta_{ch} = a_{\text{eff}} \cdot Q^2 + b_{\text{eff}} \cdot Q \tag{46}$$

Then in this fan, similarly to what happens with the compressor, the speed is controlled, and the **pressure drop** is obtained as a function of it:

$$dp = dp_{ch} \left(\frac{\omega_{\text{ref}}}{\omega} \right)^2 \cdot Dr^2 \tag{47}$$

Where ω is the actual speed and ω_{ref} is the reference speed, while Dr is the diameter scale factor.

In this model there are different interconnected equations, this one links the **volumetric flow rate**, used in the equation above, to the **mass flow rate**:

$$Q = \frac{\dot{m}_A \cdot R_A \cdot T_A}{p_A} \tag{48}$$

Then, considering that the pressure drop linked with the volumetric flow rate gives a measurement of the **fluid power**, this can be related through efficiency to the mechanical domain by the following system of equations:

$$P_{fluid} = Q \cdot dp \tag{49}$$

$$P_{brake} = \frac{P_{fluid}}{\eta_{nom}} \tag{50}$$

$$P_{brake} = T \cdot \omega \tag{51}$$

Where T is the Torque and ω is the input speed, namely the controlled variable given as input.

Equations of **conservation of mass and energy** are considered as before shown for the compressor.

Since the fan works both in the mechanical and moist air domain, there are different mass and energy components to be accounted for; anyway, although newer versions of Simscape include droplet dynamics, this had to be removed to ensure compatibility with the older version.

6.1.3 Results

The HiL simulations have been performed in a very similar way to what has been shown for the simulation with the previous model, so results can be directly shown.

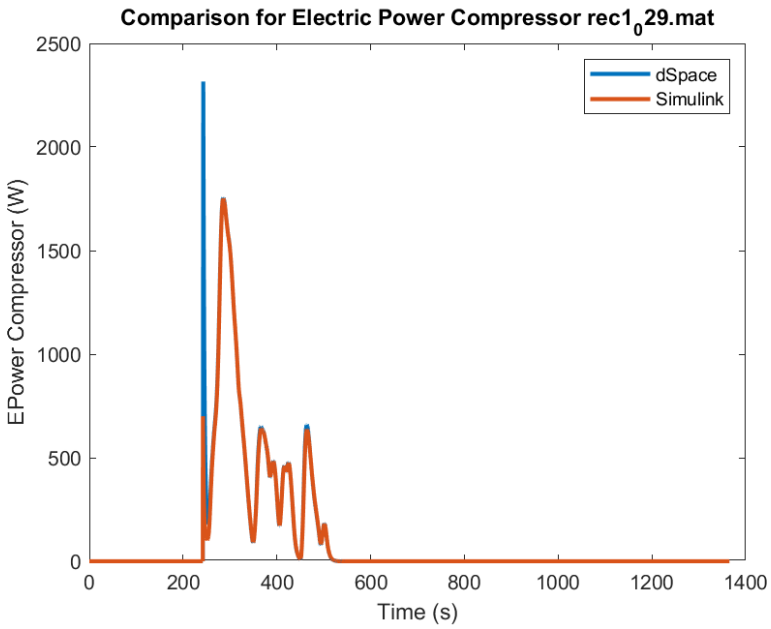


Figure 72: HiL simulation custom components: EPower Compressor

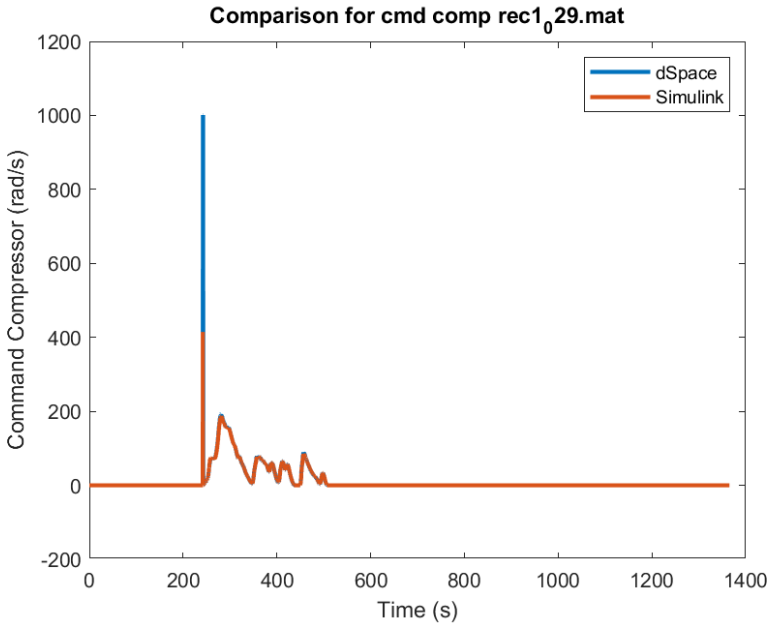


Figure 73: HiL simulation custom components: Command Compressor

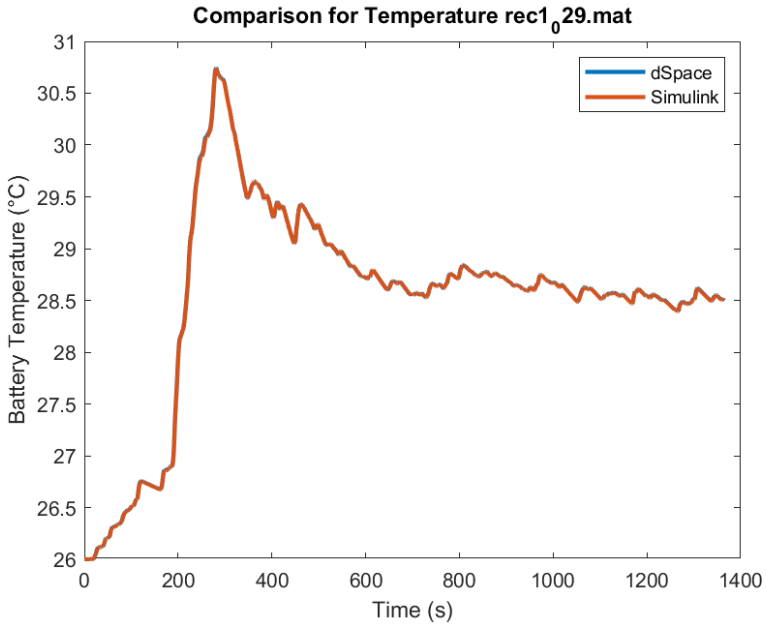


Figure 74: HiL simulation custom components: Temperature

Parametro	EPower Compressor	Command Compressor	Temperature
RMSE	62.86 W	24.44 rad/s	0.011 °C
MAE	5.51 W	0.95 rad/s	0.006 °C
NMAE	1.83%	1.59%	0.02%
MAXAE	1772.30 W	1000.97 rad/s	0.081 °C

Table 5: Results HiL with custom components

These data show an interesting behavior for the new model. It can be seen that these results have a different tendency if compared to the previous (the ones for HiL simulation are considered for similarity of Hardware); indeed, while the **RMSE** of the Electric Power of the Compressor is higher (62.89 W vs 25.74 W, Tables 4, 5), the one related to the Compressor Command is much lower for all the errors (RMSE(rad/s) 24.44 against 87.91, MAE(rad/s) 0.95 against 5.39, Tables 4, 5).
It can indeed be seen that with the new component, the response of the compressor to the command is better: the compressor's power rises

instantaneously producing a peak and this reduces the oscillations of the command as shown in Figure 73.
So errors are reduced, but as far as the Power is concerned, the RMSE is higher as mentioned before: this kind of error that considers the root of the mean squared error penalizes much more the outliers; the other errors are anyway slightly lower (**MAE**(W) 5.51 against 6.06, **NMAE** 1.83 % against 2.66 %, Tables 4, 5) because after the first peak, the values converge completely with Simulink ones.
The temperature considerations are unchanged since the system can easily achieve the **Target Temperature**.

7 Conclusions

In conclusion, this work shows the feasibility of a real-time implementation for an AMPC-based Battery Thermal Management System. The main goal was to demonstrate how a more complex Controller with predictive logic could work in an operational environment with a relatively low error on the target temperature, the object of control, even given the latencies and constraints that a HiL simulation introduces.

Some considerations can be made on each simulation:

- The first simulation, with a **single model** uploaded on Scalexio, showed that, even with overruns and errors correspondingly to the most dynamically demanding parts, the system could run in real-time in a single memory stack despite its computational complexity.
- The **double model** simulations show that, as highly expected and shown in literature [26], dividing a demanding model into two different memory stacks, enhances the processor performances by balancing computational load, more than halving the errors. Moreover, the use of an explicit solver for the Controller could be taken into serious consideration for future developments to reduce the computational load even more.
- Although the CAN bus introduces some latencies that make the error rise if compared to the single Hardware simulations, **HiL simulation** performances are similar to the first simulation, and, while the NMAE relative to the Electric Power and the Command of the Compressor are reduced to a very low amount ($< 2\%$) with the custom components, the Controller maintains very **low** the **error** of the **Target Temperature**, outperforming similar results shown in literature (NMAE 1.7352 % for the Temperature of the battery [27]) with a NMAE much lower than 1 % for all the simulations.

- The introduction of custom components reduces the oscillations and, despite an initial peak, shows better performances in terms of convergence with offline simulations.

Furthermore, for future developments, having a CAN bus payload lower than 1 %, the use of this Controller within the CAN bus already installed in the vehicle can be investigated. Moreover, given the high timestep of the MPC, a LIN Bus can be tested, to assess the performance with a cheap but effective alternative. Anyway, CAN payload should be further investigated with more advanced logging instruments to understand the real influence of latencies on the results.
The main bottlenecks in this work are related to the Plant itself, which, due to its stiffness, constrained the simulations in terms of initial conditions, MPC parameters, solver, and timestep, increasing the number of errors due to overruns.
Anyway, the main goal was to ensure the feasibility of the Controller's algorithm in a real-time environment, being the only one interested in a real-time simulation. New settings and a simplified Plant model, could lead to more testing freedom for the Controller; in this context, other solvers, even explicit, can be investigated as mentioned before.
Furthermore, the full system with HVAC could be tested to make another step forward in this research.

In the end, it is possible to conclude that the insights of this work pave the way for the application of more precise and robust Controllers with predictive behavior to a Battery Thermal Management System; this type of controller is ideal for this application: heat exchange is characterized by very slow dynamics, allowing the controller to operate with high timesteps, as in this case, ensuring an optimal behavior for a real-time simulation; its application could be key to enhancing battery performance, autonomy and life cycle.

References

[1] International Energy Agency (IEA), *Global EV Outlook 2024*, https://www.iea.org/reports/global-ev-outlook-2024/trends-in-electric-cars.

[2] Xia, G., Cao, L., Bi, G., *A Review on Battery Thermal Management in Electric Vehicle Application*, *Journal of Power Sources*, vol. 367, pp. 90-105, November 2017.

[3] Recent Advancements in Battery Thermal Management Systems for Enhanced Performance of Li-Ion Batteries: A Comprehensive Review, *MDPI Batteries*, https://www.mdpi.com/2313-0105/10/8/265,

[4] Mathworks, *Electric Vehicle Thermal Management*, https://it.Mathworks.com/help/hydro/ug/sscfluids_ev_thermal_management.html

[5] Lemort, V., Olivier, G., de Pelsemaeker, G., *Thermal Energy Management in Vehicles.*

[6] University of Stuttgart, *Model Predictive Control*, Institute for Systems Theory and Automatic Control, https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/.

[7] State-Space Representation, Massachusetts Institute of Technology, https://web.mit.edu/2.14/www/Handouts/StateSpace.pdf

[8] Mathworks, *Adaptive Model Predictive Control (MPC)*, https://it.Mathworks.com/help/mpc/ug/adaptive-mpc.html

[9] Mathworks, *Optimization Problem in Model Predictive Control (MPC)*, https://it.Mathworks.com/help/mpc/ug/optimization-problem.html

[10] Novara, C., *Driver Assistance System Design A*, lecture notes

[11] Bélanger, J., Venne, P., Paquin, J.-N., *The What, Where and Why of Real-Time Simulation*, IEEE.

[12] Mathworks, *Compare Solvers in Simulink*, https://it.Mathworks.com/help/simulink/ug/compare-solvers.html.

[13] Mathworks, *Solvers in Simulink*, https://it.Mathworks.com/help/simulink/gui/solver.html.

[14] MathWorks, *Fixed-Step Solvers in Simulink*, https://www.mathworks.com/help/simulink/ug/fixed-step-solvers-in-simulink.html

[15] LibreTexts, *Backward Euler Method*, https://math.libretexts.org/Bookshelves/Differential_Equations/Numerically_Solving_Ordinary_Differential_Equations_%28Brorson%29/01%3A_Chapters/1.03%3A_Backward_Euler_method.

[16] LibreTexts, *Newton's Method*, in *Calculus 3e (Apex)*, https://math.libretexts.org/Bookshelves/Calculus/Calculus_3e_(Apex)/04%3A_Applications_of_the_Derivative/4.01%3A_Newton's_Method,

[17] Wang, S., *Stability Analysis*, https://www2.it.uu.se/itwiki.php?page=edu/course/homepage/bridging/ht13/Stability_Analysis.pdf&action=browse.

[18] dSPACE, *dSPACE Documentation*, available under license.

[19] Böhm, S., König, H., *Real-Time-Shift: Pseudo-Real-Time Event Scheduling for the Split-Protocol-Stack Radio-in-the-Loop Emulation*, *Brandenburg University of Technology (BTU) Cottbus-Senftenberg*, 2023.

[20] *Stack Memory*, in *The Definitive Guide to the ARM Cortex-M0*, 2011, https://www.sciencedirect.com/topics/engineering/stack-memory.

[21] Diario di un Analista, *Valutazione delle prestazioni di un modello di regressione*, https://www.diariodiunanalista.it/posts/valutazione-delle-prestazioni-di-un-modello-di-regressione/.

[22] Munoz-Hernandez, G., Mansoor, S.P., Jones, D.I., *Hardware-in-the-Loop Simulation*, in *Advances in Industrial Control*, Springer, 2013, pp. 139-158, doi: 10.1007/978-1-4471-2291-3_8.

[23] Crovetti, P. S., *Electronic Systems for Vehicles*, lecture notes.

[24] CS Selectronics, *CAN DBC File Database Introduction*, https://www.csselectronics.com/pages/can-dbc-file-database-intro.

[25] Mathworks, *Creating Custom Components in Simscape*, https://it.Mathworks.com/help/simscape/lang/creating-custom-components.html.

[26] Pandit, H. R., Biju, N., Pisharodi, V., Dimitrakopoulos, P., Shenoy, M., *Framework for Digital Twin Real-Time Battery System for Model-in-the-loop and Hardware-in-the-loop Simulations*, in *2023 IEEE Transportation Electrification Conference Expo (ITEC)*, Detroit, MI, USA, 2023, pp. 1-6, doi: 10.1109/ITEC55900.2023.10187077.

[27] Kumar, P., Fuerth, C., Rankin, G., Pattipati, K. R., Balasingam, B., *Hardware in the Loop Demonstration of Battery Surface Temperature Prediction*, in *2022 IEEE International Conference on Environment and Electrical Engineering and 2022 IEEE Industrial and Commercial Power Systems Europe (EEEIC / ICPS Europe)*, Prague, Czech Republic, 2022, pp. 1-6, doi: 10.1109/EEEIC/ICPSEurope54979.2022.9854750.