Universidad Nacional
de Educación a Distancia

Escuela Técnica Superior de
Ingeniería Informática

Universidad
Complutense de Madrid

Facultad de Informática

ELECTRIC POWERTRAIN MODELING IN MODELICA

Samuel Rodrigo Rubio

Director: Alfonso Urquía Moraleda

Co-director: José Manuel Díaz Martínez

Trabajo de Fin de Máster

Máster Universitario
en Ingeniería de Sistemas y de Control

Curso 2024/2025, convocatoria ordinaria

Master's Degree Thesis
Master's Degree in Systems and Control Engineering

ELECTRIC POWERTRAIN MODELING IN MODELICA

Type B Project
Specific project proposed by student

Thesis presented by
Samuel Rodrigo Rubio

Under the supervision of

Alfonso Urquía Moraleda
José Manuel Díaz Martínez

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: Samuel Rodrigo Rubio

Abstract

In recent years, the rise of electric mobility has significantly boosted the development of electric vehicles (EVs) and their main components. Among them, the battery stands out as one of the most critical elements, as it directly affects key parameters such as range, charging speed and system lifetime.

To advance in the design and optimisation of these vehicles, it is essential to have simulation tools to analyse the energy behaviour of the system under different driving conditions. These tools reduce the need for physical prototypes and allow configurations to be evaluated quickly and at low cost.

Existing libraries and tools in simulation environments such as Modelica offer general electrical components or detailed motor models, but in many cases lack a unified, flexible framework specifically oriented to the simulation of complete EV powertrains.

This thesis presents the ***EPowertrain library***, a modular and reusable Modelica-based library designed to simulate the energy behaviour of electric powertrains under standardised and real-world driving cycles. The library includes configurable models for the main electrical components (battery, DC motor, converter, control blocks), and is focused on lithium-ion powered electric passenger cars. Its architecture is optimised for fast yet realistic simulations, aiming to support research, education, and early-stage development.

The library's performance has been validated through simulations using both the UDDS driving cycle and experimental data from real-world trips of a BMW i3. Results confirm its suitability for analysing energy consumption, comparing configurations, and evaluating the impact of control strategies in a physically consistent and computationally efficient environment.

Keywords: Modelica, Electric Vehicles, Energy Simulation, Battery Consumption, EPowertrain Library

Resumen

En los últimos años, el auge de la movilidad eléctrica ha impulsado notablemente el desarrollo de los vehículos eléctricos (VE) y de sus principales componentes. Entre ellos, destaca la batería como uno de los elementos más críticos, ya que afecta directamente a parámetros clave como la autonomía, la velocidad de carga y la vida útil del sistema.

Para avanzar en el diseño y optimización de estos vehículos, es fundamental disponer de herramientas de simulación que permitan analizar el comportamiento energético del sistema en diferentes condiciones de conducción. Estas herramientas reducen la necesidad de prototipos físicos y permiten evaluar configuraciones rápidamente y a bajo coste.

Las librerías y herramientas existentes en entornos de simulación como Modelica ofrecen componentes eléctricos generales o modelos detallados de motores, pero en muchos casos carecen de un marco unificado y flexible orientado específicamente a la simulación de cadenas cinemáticas completas de VE.

Esta tesis presenta la librería EPowertrain, una librería modular y reutilizable basada en Modelica diseñada para simular el comportamiento energético de las cadenas cinemáticas eléctricas bajo ciclos de conducción estandarizados y reales. La biblioteca incluye modelos configurables para los principales componentes eléctricos (batería, motor de corriente continua, convertidor, bloques de control), y se centra en turismos eléctricos alimentados con iones de litio. Su arquitectura está optimizada para realizar simulaciones rápidas pero realistas, con el objetivo de apoyar la investigación, la educación y el desarrollo en fases tempranas.

El rendimiento de la biblioteca se ha validado mediante simulaciones que utilizan tanto el ciclo de conducción UDDS como datos experimentales de viajes reales de un BMW i3. Los resultados confirman su idoneidad para analizar el consumo de energía, comparar configuraciones y evaluar el impacto de las estrategias de control en un entorno físicamente consistente y computacionalmente eficiente.

Palabras clave: Modelica, Vehículos Eléctricos, Simulación Energética, Consumo de Baterías, Biblioteca EPowertrain.

Contents

Figures

Tables

Glossary

0D	Zero-Dimensional Model
1D	One-Dimensional Model
1DOF	One Degree of Freedom Model
2D	Two-Dimensional Model
3D	Three-Dimensional Model
AC	Alternating Current
AMS	Analog Mixed Signal
API	Application Programming Interface
ARC	Accelerated Rate Calorimetry
AWD	All Wheels Drive
BDF	Backward Differentiation Formula
BMS	Battery Management System
CAD	Computer Assisted Design
CAE	Computer-Aided Engineering
CFD	Computational Fluid Dynamics
DAE	Differential-Algebraic Equations
DASSL	Differential Algebraic System Solver
DC	Direct Current
DFN	Doyle-Fuller-Newman Model
DLR	Deutsches Zentrum für Luft- und Raumfahrt (German Cerospace Center)
DOE	Design of Experiments
DOF	Degree of Freedom
ECU	Electronic Control Unit
EES	Electric eEnergy Storage
EHM	Electrochemical-Hydraulic Model
EMF	ElectroMagnetic Field
EMS	Energy Management Strategy
EV	Electrical Vehicle
FEM	Finite Element Model
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
FPGA	Field Programmable Gate Arrays
HIL	Hardware In the Loop
IDA	Implicit Differential Algebraic Equations System Solver
IoT	Internet of Things
MCU	Motor Control Unit
MSL	Modelica Standard Library
NEDC	New European Driving Cycle
NMPC	Non Lineal Predictive Control
OCV	Open-Circuit Voltage
ODE	Ordinary Differential Equation
OEM	Original Equipment Manufacturer
PID	Proportional Integral and Derivative Controller

PMDC	Permanent-Magnet DC Motor
PMSM	Permanent-magnet, Synchronous Motor
PWM	Pulse Width Modulation
RC	Resistance-Capacitor Model
Rint	Internal Resistance Model
ROM	Reduced Order Model
SOC	State of Charge
SPICE	Software Process Improvement Capability Determination
TR	Thermal Runaway
UDDS	Urban Dynamometer Driving Schedule
VHDL	Very High-Speed Hardware Description Language
WLTP	World Harmonized Light-duty Vehicle Test Procedure

1 Introduction, Goals and Structure

1.1 Introduction

The increasing awareness about climate change, the need to reduce greenhouse gas emissions and the search for greater energy efficiency have driven the development of alternatives to fossil-fuelled vehicles. In this context, electric vehicles (EVs) have established themselves as a viable solution for moving towards a more sustainable mobility model.

However, the energy performance of these vehicles still presents significant challenges, especially in terms of range, recharging times and battery degradation. These limitations are related to the design and operation of the electric powertrain, which is the system responsible for transforming the electrical energy stored in the battery into useful movement on the wheels.

A typical electric powertrain in a battery-powered vehicle is composed of the following main subsystems:

- A battery (in this case lithium-ion), which acts as the energy source.
- A power converter (DC-DC or inverter), which adapts the voltage and current required for the motor.
- An electric motor (DC or AC), which converts electrical energy into mechanical energy.
- And the corresponding control systems, which manage traction, regenerative braking and overall system efficiency.

Compared to internal combustion engine (ICE) vehicles, EVs have an architecture that is mechanically simpler but much more dependent on power electronics and optimised energy management. The overall system efficiency therefore depends on multiple interrelated factors, such as control strategy, powertrain topology and driving conditions.

To evaluate and optimise these systems, simulation tools play a key role. They allow exploring alternative configurations, validating control algorithms and estimating energy consumption under standardised (such as WLTP or UDDS) or real driving cycles, all without the costs associated with building physical prototypes.

However, the existing libraries in Modelica, although powerful, often offer isolated or too general component models, without a structured and modular framework specifically oriented to the energy analysis of complete electric vehicle powertrains.

This project responds to this need by developing ***EPowertrain***, a modular and reusable library implemented in Modelica, designed to simulate the energy behaviour of an electric vehicle of the passenger car type, powered by a lithium-ion battery. The library has been designed to facilitate the design, validation, and comparison of propulsion architectures from a flexible, reproducible and physically coherent approach.

1.2 Goals

The main objective of this work is the development of a modular and reusable library in Modelica, oriented to the simulation of the energy consumption of battery electric vehicles. The library, called *EPowertrain*, has been designed to faithfully represent the interactions between the main components of the electric powertrain, maintaining a level of complexity suitable for efficient and flexible simulations.

Through this library, the purpose is to facilitate the analysis of alternative propulsion architectures, as well as the evaluation of control strategies under realistic driving conditions. The specific objectives of the work can described as follows:

- **Develop modular models** of the main electrical components of the system: battery, motor, power converter and control interfaces.

- **To ensure the reusability and extensibility** of the models by means of an object-oriented structure and acausal connectors, following the clever design practices in Modelica.

- **Simulate the energy behaviour of the system** under different driving profiles, including standardised cycles such as UDDS [1] and real data obtained from urban journeys [2].

- **Validate simulation results against experimental data**, analysing the accuracy of the library in the estimation of key variables such as energy consumption, torque or battery state of charge (SOC).

To maintain the focus on the analysis of energy consumption, certain aspects have been left out of the scope of the project that, although relevant in other contexts, are not essential for the defined objectives:

- **Detailed thermal Modeling:** The thermal domain and thermal management systems are not included in order to avoid unnecessary complexity in simulations focused on power consumption.

- **Battery degradation:** Long-term evolution of internal capacity or resistance is not considered, as this is a cumulative phenomenon outside the framework of individual cycles.

- **Advanced longitudinal vehicle dynamics:** A simplified equivalent load model has been adopted, without including suspensions, mass transfers or detailed tyre models.

These decisions have allowed work to focus on the electrical representation of the system, optimising the fidelity and computational performance of the simulations. The *EPowertrain* library thus aims to be a useful educational and technical tool for the exploration, validation and comparison of electric vehicle configurations from an energy perspective.

1.3 Document Structure

This document is structured in seven chapters, organised in a progressive way for the development of the work from its motivation to the experimental validation of the implemented models:

- **Chapter 1** introduces the motivation for the work, the objectives set, and the scoping decisions taken. It also describes the general characteristics of the library developed.

- **Chapter 2** presents a review of the state of the art in electric vehicle Modeling, with special emphasis on the use of Modelica and its most relevant libraries. Different battery and powertrain Modeling approaches are also discussed, as well as fidelity levels and validation strategies present in the literature. Finally, compares Modelica with other simulation tools commonly used in the electric vehicle industry, evaluating their physical Modeling capabilities, computational performance and integration with open standards.

- **Chapter 3** focuses on the conceptual and mathematical description of the system to be modelled. A functional decomposition of the electric powertrain into subsystems (battery, converter, motor, chassis) is presented, detailing the differential and algebraic equations governing their behaviour. Modeling assumptions, definitions of symbols and units are presented, and the selection of simplified models is justified to ensure computational efficiency.

- **Chapter 4** describes the internal structure of the developed library, explaining the function of each subpackage (Interfaces, Electrical, Mechanical, Control, Sensors, etc.). The design decisions, the connections between components and the modularity strategies adopted are detailed. Diagrams, icons and graphical annotations taken directly from the Modelica environment are included to facilitate the understanding and implementation of the mathematical models presented in chapter 3.

- **Chapter 5** deals with the validation of the complete system by means of driving cycles, both standardised (UDDS) and real (BMW i3), analysing the agreement of the model with the measured data and justifying the observed deviations.

- **Chapter 6** presents the conclusions of the work, as well as a proposal of future lines for the extension and improvement of the library.

- **Appendix A** contains the complete list of the Modelica code that makes up the developed library. It is structured according to the functional order of the

subpackages and is accompanied by brief descriptions to facilitate its navigation. This annex allows the full reproducibility of the models and their reuse in future projects.

- **Appendix B** contains the self-generated documentation from the Dymola environment for the *EPowertrain* library. It includes a hierarchical representation of the packages and models, as well as the descriptions, annotations and iconographic diagrams defined in each class. This documentation provides a global view of the functional structure of the library, facilitating its understanding, maintenance and reuse.

2 State of the Art

2.1 Introduction

In the context of the transition towards electromobility, the analysis and optimisation of the energy performance of electric vehicles (EVs) is essential to improve their range, charging efficiency and commercial viability. However, direct experimentation with physical prototypes has significant limitations in terms of cost, time and flexibility. Therefore, Modeling and simulation of physical systems have become fundamental tools in EV design and validation, allowing the virtual exploration of multiple configurations and operational scenarios without the need to build numerous prototypes [3]. In fact, the practice of virtual validation (e.g. using digital twins) is becoming increasingly widespread in the automotive industry, as it significantly reduces the costs associated with prototyping and speeds up the development cycle.

An electric vehicle involves the interaction of sub-systems of different physical nature - mechanical, electrical, electronic, thermal, etc. - whose integrated behaviour determines the overall performance of the vehicle. The complexity of these multi-domain systems makes it necessary to have simulation environments that allow Modeling components from different physical disciplines in a coupled way, reproducing phenomena such as electro-mechanical conversion in the engine, battery power delivery under different conditions, or vehicle dynamics in response to load profiles. Having sufficiently accurate computational models of each subsystem, integrated on a common platform, enables advanced approaches to validate and optimise electric powertrains prior to the construction of real prototypes. For example, simulators are extensively used to develop optimal energy management strategies in hybrid and electric vehicles (e.g. predictive control), taking advantage of the increasing computational capacity to optimise the coordinated use of the vehicle's different power sources. The need for physical Modeling tools in the field of EVs is justified by:

- The possibility to study vehicle energy and dynamic behaviour under a multitude of conditions without incurring the costs and time of physical prototyping,

- Capacity to evaluate novel component configurations and control strategies in a safe and reproducible environment.

- Multi-disciplinary nature of EVs, which requires a multi-domain simulation approach to capture the interdependencies between electrical, mechanical and control components.

The following sections detail the fundamentals of acausal, multi-domain physical Modeling (section 2.2), the characteristics of the Modelica language as a leading environment for this purpose (2.3), the main alternative simulation tools used in industry (2.4) and, finally, a comparison from the literature between Modelica and these alternatives in the context of EV Modeling (2.5).

2.2 Multi-domain Physical Modeling in Acausal Environments

Multidomain physics Modeling consists of representing complex systems spanning multiple domains of physics (electrical, mechanical, thermal, hydraulic, etc.) under a unified framework of equations. A modern approach to achieve this is equation-based acausal environments, in contrast to traditional block diagram (causal) tools. In an acausal environment, model components are defined by mathematical relationships (algebraic and differential equations) that describe their behaviour, without predetermining the direction of information flow (input/output) between them. This means that the connections between components represent bidirectional physical interactions (e.g., an electrical connector imposes equal voltage and conserves current, or a mechanical joint balances forces and accelerations between connected bodies) rather than a unidirectional signal. The result is a declarative model, where the user describes which physical relationships govern the system, leaving it to the solver to determine how causal dependencies propagate during the simulation [4].

Example: Consider an ideal electrical resistor. In acausal language, its behaviour can be specified by Ohm's law declaratively:

$$V = R \cdot I \tag{2.1}$$

where V is the voltage drop across the resistor, R it is the resistance and I the current through it. This single equation works whether the resistor is connected to a voltage source (determining a current) or a current source (determining a voltage), as the system solver will calculate the appropriate causality in each context. In causal environments (e.g. Simulink), on the other hand, it would be necessary to define different blocks or configure the resistor block in different modes depending on whether it is excited with a current or a voltage, as it is required to fix a priori which variable is input and which is output. Acausal Modeling avoids this drawback by not fixing the direction of the relationships, providing great flexibility and reusability of components in different scenarios [5].

Multidomain acausal Modeling environments (such as Modelica, see section 2.3) are mathematically based on systems of differential-algebraic equations (DAEs). A complete model is formed by assembling elementary components (each with its internal equations) through connectors that impose continuity constraints (e.g., equal electric potential at a common node, equal velocity in a mechanical coupling) and conservation laws (e.g., zero sum of currents at a node, zero sum of forces at a static junction).

The symbolic environment solver gathers all the equations of the system and applies analysis methods (e.g. index reduction, selection of state variables, etc.) to prepare a numerically solvable system. Finally, a numerical integrator (e.g., explicit or implicit integration methods) solves the equations in time. This automatic chain (translation of models to DAEs and numerical solution) frees the modeller from having to manually derive the causal forms of each equation, allowing to focus on the physics of the problem.

Acausal Modeling offers several advantages:

- **Modularity:** it is possible to freely drag and connect component instances in almost any physically valid configuration, without the need to manually adapt interfaces, as acausal connectors ensure physical compatibility (e.g. freely connecting electrical elements by common nodes, mechanical elements by flanges, etc.) [6].

- **Reusability and modularity:** the same sub-model (e.g. an electric motor) can be used at different system levels or in different projects without modification, as it does not carry signal direction assumptions; furthermore, thanks to inheritance and parameterisation support, basic models can be extended to create specialised variants without rewriting from scratch [5, 7].

- **Multi-domain capability**: by relying on generic equations, acausal languages can represent electrical, mechanical, thermal, etc. components and their interactions on a single platform, avoiding fragmentation into multiple tools. This facilitates comprehensive studies of complex systems such as EVs, where the battery (electrical/chemical), motor (electrical/mechanical) and electronic control (digital/algorithmic) must be evaluated together [8].

- **Flexibility of configuration changes:** replacing one component with another (e.g. a detailed motor model with a simplified map-based one) does not require redoing connections and interfaces, as long as they share the same connector type, which speeds up the exploration of alternative designs in multiple tools. This facilitates comprehensive studies of complex systems such as EVs, where the battery (electrical/chemical), motor (electrical/mechanical) and electronic control (digital/algorithmic) must be evaluated together [9].

Despite its advantages**, the acausal approach comes with some challenges**. One of them is the **longer learning curve**: engineers must familiarise themselves with the equation-based paradigm and concepts of DAEs, which differs from the sequential signal flow they might be used to with causal tools. For example, it has been observed that Modelica tools such as Dymola require a higher initial learning effort compared to environments such as Simulink [10], in exchange for greater customisation and expressive. This flexibility can be overwhelming for novice users, although access to libraries of predefined components and didactic examples mitigates the problem [5, 6].

Another aspect is **model debugging**: since the source code is declarative equations, when an error occurs (e.g., redundant or missing equations that make the system overdetermined or indeterminate), diagnosis may be less intuitive than in causal schemes. However, modern environments often provide symbolic debugging tools and messages that guide the user in the correction (e.g., indicating a variable without an associated equation, suggestions to provide initial conditions, etc.). In terms of simulation performance, while acausal engines allow efficient simulations in many cases, certain very detailed models (e.g., high frequency switched circuits, or systems with very frequent discontinuous events) may require very small integration steps or special techniques, similar to other platforms [5]. Thus, averaging models or order reductions are sometimes used to tractably simulate fast phenomena in long-term studies [11, 12].

In summary, multi-domain acausal environments provide a powerful and general framework for physical Modeling, with great benefits in versatility and model accuracy, while requiring the user to be properly trained in the fundamentals of equation Modeling and to use the available libraries correctly.

Table 2.1: Comparison Between Acausal and Causal Modeling Approaches.

	Acausal Modeling	**Causal Modeling**
Paradigm	Based on algebraic and differential equations (DAEs), with no predefined direction of data flow.	Based on block diagrams with explicitly defined inputs and outputs (cause-effect relations).
Flow direction	Determined automatically by the solver at simulation time.	Manually defined by the user for each block.
Component connectivity	Bidirectional; represents shared physical variables (e.g., voltage, force, torque).	Unidirectional; signals are propagated from output to input.
Physical representation	Closer to actual physical formulation (laws of physics expressed as equations).	Requires transforming physical laws into signal-based structures.
Model reusability	High; components can be reused in multiple contexts without rewriting equations.	Limited; models often need to be adapted to each signal flow configuration.
Modularity and composability	High; components can be interchanged if physical connectors are compatible.	Lower; changes in model structure often require reworking block connections.
Typical environments	Modelica, Simscape, Simcenter Amesim.	Simulink, LabVIEW, classic control systems.
Learning curve	Longer, requires understanding of physical Modeling and DAEs.	Faster, intuitive for users with signal-flow or control background.
Typical applications	Physical Modeling of electrical, mechanical, thermal, and hybrid systems.	Control design, signal processing, process simulation.

2.3 Relevant Modelica Libraries

Developed since the late 1990s by the Modelica Association [13], Modelica has established itself as one of the most widely used multi-domain causal languages in academia and industry. Modelica is an equation-based, object-oriented language for Modeling complex physical systems [14].

Its philosophy is based on describing the behaviour of components by means of mathematical relationships (ordinary algebraic-differential equations) rather than sequential procedures, allowing a declarative representation of system dynamics. The

language also supports discrete events, allowing the simulation of hybrid (continuous-discrete) systems to include control logics, condition-triggering, etc. The language specification is open and maintained by the Modelica Association, which also provides a comprehensive Modelica Standard Library (MSL) [15] with fundamental models in numerous domains: electrical (analogue and digital), translational and rotational mechanical, thermal, hydraulic, among others. Thanks to this standard library and many other available libraries, modellers can build virtual prototypes of a wide variety of physical systems by combining predefined components and adding their own equations when necessary.

In Modelica, each model is essentially a class that encapsulates equations and variables that can be reused as a subcomponent in higher-level models. The language fully supports the principles of modularity and inheritance: one model can extend (inherit) from another by adding or modifying equations/parameters, facilitating the creation of specialised variants without duplicating code. Modelica also allows acausal connectors to be defined for different domains (e.g. electrical pin connectors with voltage and current variables, mechanical flange connectors with position and force, etc.), allowing components to be graphically connected in a physically meaningful way. During compilation, the Modelica engine gathers all the equations of the connected components and generates the overall system to be solved. This ability to bring together multiple formalisms under a unified syntax makes Modelica a particularly suitable tool for complex systems such as electric vehicles, where heterogeneous phenomena coexist.

Modelica libraries play a fundamental role in the process of Modeling and simulating complex physical systems. These libraries group sets of parameterizable models of basic components -such as resistors, electric motors, batteries, converters or mechanical elements- that can be reused and combined to build complete systems [1,2]. Their main functions include:

- **Facilitating reusability and modularity**: Libraries allow complex models to be built from validated blocks, reducing development time and increasing the robustness of simulations.
- **Promoting physical consistency**: By using standardised connectors (based on stress and flow variables), it is guaranteed that the interaction between subsystems respects physical principles such as energy conservation.
- **Speed up validation and comparison**: By relying on previously tested components, it facilitates the validation of new models and speeds up the comparison between different configurations.
- **Encourage extensibility**: Many libraries are designed in an open way, allowing users to extend or specialise models to suit specific needs.

Modelica has been widely used in academic research on electric vehicles, ranging from single component studies to complete system optimisation. For example, Qin et al. [12] modelled a lithium-ion battery pack with a second-order equivalent circuit scheme in Modelica, successfully simulating its dynamic behaviour inside an electric vehicle and validating it against experimental data.

Bui et al. [16] used Modelica to develop and evaluate in real time an energy management strategy for a hybrid vessel, demonstrating the language's ability to integrate battery models, motors and control algorithms in a modular way. Another

example is provided by Milishchuk and Bogodorova [17], who implemented a Thevenin-type battery model in Modelica incorporating ageing (degradation) effects, and reported the effectiveness of this approach to study the evolution of battery performance with use.

These cases, together with numerous contributions in specialised conferences (e.g. International Modelica Conference, IEEE Vehicular Technology, etc.), show the maturity of Modelica as a reference platform for electric vehicle research.

Finally, it is worth mentioning that the Modelica ecosystem is supported by multiple simulation environments. Among the commercial environments, Dymola (Dassault Systemes) has historically been the most widely used, offering a robust Modeling and simulation environment with Modelica and powerful analysis tools (optimisation, linearisation, etc.). There are also open-source tools such as OpenModelica, which implements the Modelica standard and allows models to be simulated free of charge, encouraging its use in academia.

This wide adoption by different vendors confirms Modelica's status as a standard language for system-level physical Modeling. Within the Modelica ecosystem, several libraries are relevant for the simulation of electric powertrains:

- **Modelica Standard Library (MSL):** The Modelica Standard Library provides a basic collection of models in multiple domains (electrical, mechanical, thermal, etc.), including ideal electric motors, simple batteries, transmission elements and control components. Although the MSL provides the basis for starting the Modeling of an EV, it does not by itself cover all specific aspects of a complete electric powertrain. Even so, the library’s open and extensively validated character makes it the starting point for many specialised developments.

In fact, the MSL has served as a foundation for vehicle-focused extensions, incorporating models of electrical machines (induction, synchronous, etc.), friction losses or thermal effects in power components. In Figure 2.1 illustrates the constitution of a DC motor by using blocks from the standard Modelica standard library.

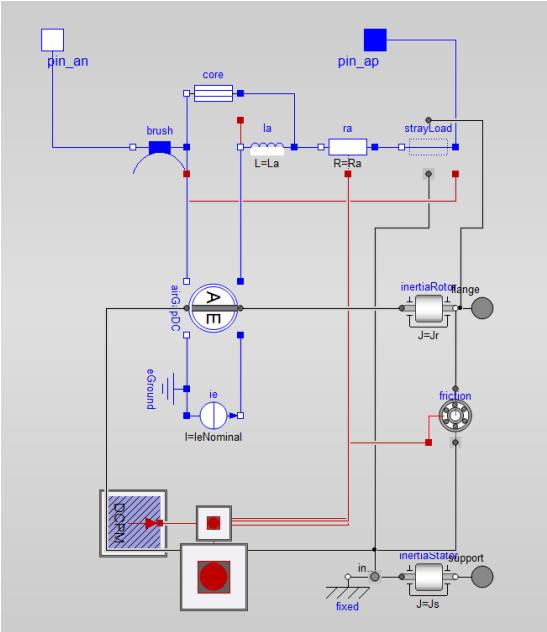

Figure 2.1: Permanent Magnet, DC Machine MSL Model.

- **VehicleInterfaces Library**: This is an architecture library that defines a standardised framework for vehicle interfaces. It provides generic templates and connectors to assemble subsystems (powertrain, chassis, driver, etc.) in a consistent way. Although the *VehicleInterfaces* library itself does not provide detailed models, its importance lies in facilitating the integration of components from different libraries under a unified vehicle schema [7]. Many works have adopted these common interfaces to build electric vehicle models by easily interchanging components (e.g. different motor or battery types).

- **Electric propulsion specific librarie**s: In response to the need for more detailed and efficient models to simulate EVs, several authors have developed specialised Modelica libraries for electric powertrains. Ceraolo presented one of the first free libraries for electric and hybrid vehicles [5], incorporating adjustable physical models of motors (synchronous and asynchronous machines), generators, converters and batteries. A key feature of this library is the use of averaged models in the power electronics components, forgoing the reproduction of high frequency switching to achieve more efficient simulations in long duration scenarios (e.g. the standard NEDC cycle of ~1200 s). In this way, a compromise between detail and computational cost was achieved that was adequate to evaluate EV dynamics over full driving cycles. The library also includes battery models and aerodynamic resistances and has served as a basis for building pure electric and hybrid EV examples in Dymola and OpenModelica.

- **ElectricDrives (Modelon):** There are commercial libraries, such as Modelon's *ElectricDrives*, focused on Modeling electric drives (DC motors, induction motors, PMSM, converters, basic controllers, etc.). These libraries provide detailed component-level models and multiple electric drive configurations. However, their scope tends to focus on the electrical behaviour of the motor and its inverter, and they do not explicitly address the energy integration of the whole vehicle

under different driving profiles. In other words, they offer an excellent level of fidelity in the simulation of a single inverter-drive train, but aspects such as energy management at vehicle level (battery operation strategies, regeneration under braking, etc.) or the influence of driving are outside their direct scope.

In addition to the above, other relevant libraries can be mentioned: for example, the *AlternativeVehicles* (DLR) library [18], of a commercial type, oriented towards hybrid, electric and fuel cell vehicles. In 2011, the *Electric Energy Storage* (EES) library was proposed [19], with battery models from cells to complete packs, a precursor to later developments incorporated in the MSL. In general, the availability of specific libraries has grown to cover distinct levels of detail according to simulation needs: from fast models for range estimation to complex thermo-electrochemical models for degradation analysis or design of advanced control strategies.

The proper use of these libraries, combined with the development of specific components, when necessary, constitutes an essential pillar for simulation projects oriented to the analysis of energy consumption in electric vehicles, such as the one proposed in this thesis.

2.3.1 Electric Powertrain Modeling

Powertrain Modeling is a central aspect in the simulation of electric vehicles, since it directly conditions the estimation of energy efficiency, autonomy and interaction between subsystems. In the Modelica environment, this process is addressed through modular architectures that allow the hierarchical and reusable representation of each functional component: from the battery to the wheels. This structure facilitates the scalability of the model and its adaptation to diverse levels of fidelity according to the analysis objectives, from fast quasi-real-time simulations to detailed studies of dynamic behaviour.

Authors such as Ceraolo [5] have developed pioneering libraries in the representation of electric and hybrid vehicles, introducing physical models with adjustable parameterization to represent generators, motors, converters and batteries. These libraries are built with an acausal approach, which allows defining the physical equations without the need to explicitly fix the directions of power or signal flow, thus favouring flexibility in the coupling between components.

In more recent work, Liu et al [20] implemented a simulation framework oriented to the analysis of energy control strategies, combining electrical, mechanical and thermal components within a Modelica environment. Using tools such as Dymola, their library allows configuring different powertrain topologies (single drive, dual drive, parallel or series hybrid), with the possibility of adjusting the complexity levels depending on the required detail. This modularity is particularly useful for applications such as virtual testing of drive cycles, optimization of acceleration profiles, or validation against experimental data. A major advantage of Modelica Modeling is the possibility of defining various levels of abstraction:

- Simplified 0D or 1D type models, suitable for rapid simulations and high-level prototyping.

- Intermediate physical models, which include nonlinear equations representative of dynamic behaviour (for example, models of losses in electric motors or transient response of converters).

- Detailed multiscale models, capable of representing thermal, electromagnetic or electrochemical phenomena more accurately, although at the cost of a higher computational load.

These methodologies are also supported by libraries or proprietary developments from manufacturers and academic institutions. The reuse and parameterization of blocks facilitates not only the Modeling, but also the calibration and validation of complete propulsion systems under variable conditions, allowing the easy integration of sensors, controllers and user models.

One of the most relevant contributions of the study is the treatment of co-simulation as the structuring axis of the complete vehicle model. Through standards such as FMI (Functional Mock-up Interface) and tools such as *Dymola*, *Simulink*, or *GT-SUITE* the authors show that it is possible to integrate subsystems developed on different platforms without compromising the fidelity of the overall system.

Different integration configurations (model exchange, standalone co-simulation, coupling per server) are exemplified, each with advantages and limitations in terms of simulation speed, model transparency and inter-tool compatibility.

Also, special emphasis is placed on the need to use fast 1D models to speed up simulation times in optimization contexts, without sacrificing the ability to capture relevant transient effects.

2.3.2 Battery Modeling

The battery represents the energy core of an electric vehicle, the Modeling of which directly influences range estimation, energy management strategies and thermal control. There are multiple methodologies to simulate its behaviour, differentiated by the level of fidelity required and the physical phenomenon of interest (electrical, thermal or electrochemical). In the field of electric vehicle simulation in Modelica, three predominant approaches can be found, each with advantages and limitations depending on the use case.

Simplified Electrical Models
Electrical models based on equivalent circuits, such as Rint, RC or Thevenin schemes, are widely used in the simulation of batteries for electric vehicles due to their low computational cost, ease of implementation and compatibility with multidomain simulation environments. These models represent the dynamic behaviour of the battery through combinations of resistors, capacitors and controlled voltage sources, which allows capturing effects such as the voltage drop associated with the internal resistance or the transient response to load changes.

However, the fidelity of these representations can vary depending on the order of the model and the phenomena considered. In this context, Qin et al [12] propose a relevant

evolution using a third-order RC model that explicitly incorporates the voltage hysteresis observed in $LiFePO_4$ cells. This structure allows capturing the internal overpotential and simulating with high accuracy both the dynamic response and the behavior in real cycles such as UDDS or NEDC, with errors lower than 2%. Implemented in Modelica using the MWorks tool, the model is easily integrated into complete electric vehicle simulations, demonstrating its usefulness in both validation and SOC estimation.

Complementarily, Chen and Rincón-Mora [21] develop a comprehensive electrical model oriented to the accurate prediction of runtime and I-V performance of batteries in portable electronics. Their proposal combines elements of Thevenin, impedance and usable capacity models, introducing a dual RC network that simulates two differentiated time constants (short and long), together with a SOC-dependent voltage source to reflect the open-circuit voltage (OCV) nonlinearity. Although its implementation is performed in the Cadence environment, the underlying principles-modularity, parameterization, and transient response are extrapolable to Modelica architectures.

Both approaches reflect a common trend: the refinement of RC models to integrate previously ignored phenomena, such as hysteresis or state-of-charge and temperature-dependent variations, without incurring the computational complexity of full electrochemical models. Thanks to their modular and parameterizable structure, these models can be implemented in libraries or proprietary developments, being particularly suitable for autonomy studies, on-board control and validation of BMS algorithms in hardware-in-the-loop (HIL) simulations.

Thermal and Heat Propagation Modeling

The thermal behaviour of lithium-ion batteries has become a critical aspect of Modeling, especially in the face of increasing energy density, fast charging demands, and the need to ensure system operational safety. During normal electric vehicle operation, the battery generates heat because of ohmic losses, non-reversible electrochemical reactions and hysteresis effects. If this heat is not properly managed, it can cause significant temperature gradients within the battery pack, accelerating degradation phenomena and even triggering thermally hazardous events such as thermal runaway (TR). The increasing energy density and the need for ultra-fast loading have intensified the interest in predictive strategies capable of anticipating these events and assessing the effectiveness of containment measures.

Thermal models aim to represent both the internal generation of heat and its propagation and dissipation to the surroundings, considering the geometrical design of the package, the thermal connectivity between cells, and the cooling systems (air, liquid or phase change materials). At the computational level, these dynamics can be addressed by 1D, 2D or pseudo-3D formulations, depending on the balance between accuracy and computational cost.

An interesting contribution in this field is the *BatterySafety* library, developed in Modelica by Groß and Golubkov [22]. It includes a simplified but efficient model for simulating TR propagation in battery packs. Based on experimental data obtained by accelerated rate calorimetry (ARC), the model employs a so-called simple tracing approach, which allows predicting the temperature evolution during the exothermic

reaction without requiring curve fitting. This approach facilitates the simulation of critical events at the cell, module and pack scale, allowing both heat generation and heat transfer between thermally connected cells to be modelled. The model incorporates variable thermal resistances that decrease in value when a cell enters TR, simulating thermal propagation driven by the release of hot gases.

From the simulation perspective in Modelica, thermal Modeling can be addressed through the *Modelica.Thermal.HeatTransfer* [23], *ThermoPower* [24] libraries, or by developing specific submodules that couple the thermal balance to temperature-dependent electrical parameters. One of the most widespread applications consists in Modeling the internally generated heat by means of:

$$Q_{gen} = I^2 \cdot R_s + I \cdot (V_{out} - V_{The}) = I^2 \cdot Z_{The} \tag{2.2}$$

Where the first term represents Joule losses and the second term represents polarisation losses on a 1RC Thevenin model [21]. This heat can then be coupled to 1D or 3D thermal models that simulate dissipation through the package, cooling modules and environment. TR propagation models allow simulating failure scenarios, evaluating passive safety strategies (insulators, compartmentalisation) and designing active countermeasures (cooling, thermal fuses). Moreover, their integration in pack-level simulations enables sensitivity analysis and validation against experimental data without resorting to expensive CFD models.

Simplified Electrochemical Models

Electrochemical models provide a detailed description of the internal phenomena of the cell, including electrolyte dynamics, charge and discharge reactions, and ion migration through the separator. However, full models - such as those based on the Nernst-Planck equation [25], Fick diffusion [26] or the Doyle-Fuller-Newman (DFN) model [27] - involve a high computational burden due to the solution of coupled partial differential equation systems, which limits their applicability in simulations of complete electric vehicle systems, especially under optimisation or co-simulation conditions.

In this context, simplified electrochemical models have emerged that allow capturing the main physic-chemical effects of interest - such as internal polarisation, potential unbalance between electrodes, and thermal dependence of capacitance - without the need for fine spatial discretisation or three-dimensional grids. These models strike a balance between realism and efficiency, making them ideal candidates for integration into energy management strategy (EMS) simulations, thermal control or validation under aggressive driving cycles.

A representative example of this approach is the work of Romero and Angerer [8], who present a formulation based on an equivalent electrochemical-hydraulic model (EHM) implemented in Modelica to simulate fast-charge dynamics under thermal and electrochemical constraints at the cell and pack level. This model considers two main states per electrode (bulk and surface concentration), allowing to simulate phenomena such as lithium plating, voltage drop associated with reaction kinetics and heat generation. The formulation includes dependencies on temperature and charge transfer parameters and is implemented in a non-linear predictive control (NMPC) environment using JModelica.org.

One of the key contributions of the study is the use of the EHM model within an optimal control scheme to minimise charging time while keeping internal variables (such as anode potential and cell temperature) within safe limits. Through simulations under fast charge cycles (up to 2C), it is demonstrated that the model is able to accurately reproduce the thermal and electrochemical behaviour of 21700 cells with immersion cooling, allowing the design of charge profiles that reduce the charge time of a full pack to 36 minutes without compromising operational safety.

2.4 Alternative Tools

The rise of the electric vehicle (EV) has pushed the need for accurate models of batteries and electric powertrains (motors, inverters, transmissions, etc.) for performance, efficiency and ageing analysis. Several multi-domain simulation tools allow these models to be built, each with different approaches and capabilities. In this chapter we analyse Modelica (as a language and ecosystem of tools, e.g. Dymola, OpenModelica, etc.) in comparison with its main alternatives widely used in automotive: MATLAB/Simulink (with the physical extension Simscape), ANSYS Twin Builder, GT-SUITE, Simcenter Amesim, among others.

Several key aspects will be evaluated and compared: the physical Modeling capabilities (multi-domain, use of acausal vs. causal equations, support of different physical domains), numerical accuracy and robustness of the simulation, compatibility and integration with other environments (including FMI/FMU standards and co-simulation), usability and learning curve, availability of specialised libraries (in particular for batteries and electric powertrain components), support of open standards and model reuse, as well as computational performance and scalability for large models.

While Modelica offers a particularly powerful approach to physical EV Modeling, there are several simulation tools and environments widely used in industry and research. In the following, some of the most relevant alternatives are briefly described - highlighting their approaches and capabilities - and compared to Modelica from an EV Modeling perspective.

2.4.1 Simulation Tools Considered

- **Modelica** - An acausal, object-oriented language for multi-domain physical Modeling. It is an open standard maintained by the Modelica Association, with commercial (e.g. Dassault Systemes Dymola, Wolfram SystemModeler, MapleSim) and open source (OpenModelica, JModelica.org, etc.) implementations. Modelica makes it possible to describe complex systems using differential-algebraic equations (DAE) instead of causal block diagrams, which facilitates the integration of different physical domains (electrical, mechanical, thermal, etc.) in a single model. It has the free Modelica Standard Library and numerous additional libraries (free to use or commercial) for different sectors (from electrical systems to complete vehicles).

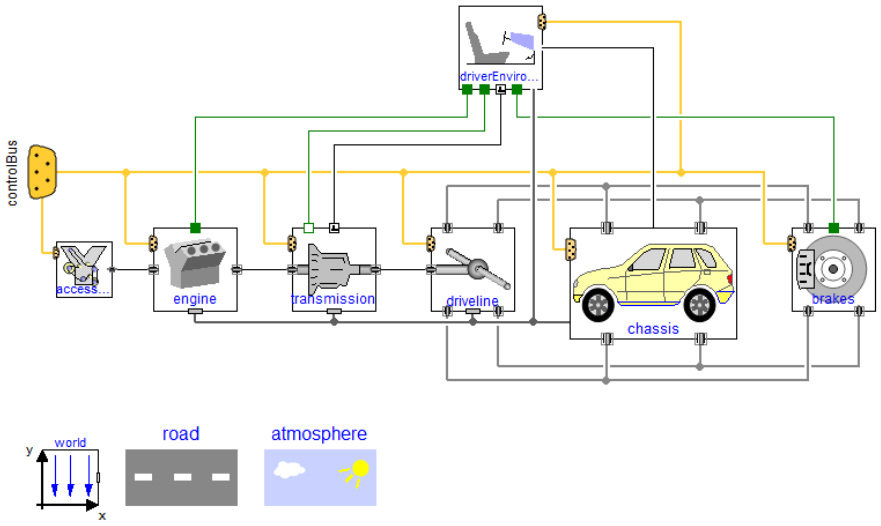

Figure 2.2: Modelica's VehicleInterfaces Library [6].

- **MATLAB/Simulink + Simscape** - MathWorks Simulink is a causal block diagram (one-way signal) based environment widely used in control engineering. For physical Modeling, Simulink is complemented by Simscape, a set of libraries and an infrastructure for acausal Modeling of physical networks (e.g., electrical circuits, hydraulic or mechanical networks). Simscape provides predefined components for multiple domains (Simscape Electrical for electrical and power electronics systems, Simscape Driveline for mechanical transmissions, Simscape Fluids for fluids and thermal, Simscape Multibody for 3D, etc.), which are integrated into Simulink diagrams. This makes it possible to combine physical plant models with control systems in the same environment. Simscape uses its own internal equations (the user can create custom components in Simscape language), solving connections in an analogous way to Modelica (energy balance, Kirchhoff, etc.), but under the hood it remains tied to the MATLAB environment.

- **ANSYS Twin Builder** - ANSYS digital twin and system simulation platform. Twin Builder (formerly Simplorer) supports hybrid multi-domain Modeling, combining acausal and causal models. In particular, it allows the use of different standard languages: from Modelica and VHDL-AMS (acausal languages of conservative components) to SPICE circuit languages, causal functional blocks and even C/C++ or Python code. Includes own libraries for power electronics, fluid-thermal systems and an integrated Modelica library with EV specific components, e.g. battery cell templates (equivalent circuits dependent on SOC, temperature, etc.), thermal management (Heating & Cooling) and electrical powertrain (motors, converters) libraries. It is especially oriented to the creation of digital twins, integrating 0D/1D models with 3D simulation (via co-simulation or reduced models) and facilitating the connection with IoT or control platforms.

- **GT-SUITE** - Multi-physics 1D simulation suite from Gamma Technologies, widely used in the automotive industry. It was born focused on internal

combustion engine Modeling (GT-Power for thermodynamic cycle) but evolved to cover all vehicle systems (air/fuel flows, aftertreatment, battery thermal systems, electrical machines, transmissions, air conditioning systems, etc.). GT-SUITE offers pre-built component libraries with a high level of fidelity (e.g. detailed models of turbochargers, intercoolers, or electric motor maps). It uses a flow-port connected component Modeling approach (like an acausal mass/energy balance scheme), although the tool takes care of resolving causality directions internally. It is highly optimised for automotive applications and supports open/closed loop simulation, parametric optimisation and real-time execution (many manufacturers use it for HIL on engine and vehicle test benches). GT-SUITE can also be integrated with Simulink (e.g. via S-Functions) and supports standards such as FMI for model exchange.

- **Simcenter Amesim** - Siemens' multi-domain 0D/1D simulation environment (originally LMS Amesim). Allows Modeling of complex mechatronic systems through a graphical drag-and-drop interface of components from a wide variety of physical libraries (electrical, mechanical, hydraulic, pneumatic, thermal, etc.). Components are connected by ports representing stress/flow variables (e.g. voltage-current, pressure-flow); connecting one component to another automatically establishes the necessary causality relationships (inputs/outputs).

  Amesim is characterised by its extensive catalogue of validated and ready-to-use components (valves, cylinders, pumps, exchangers, electric motors, converters, etc.), which speeds up the construction of industrial models. Additionally, it supports the Modelica language within its components, allowing the incorporation of custom models in Modelica or the reuse of non-native Modelica libraries from Amesim. It is a tool recognised for its ease of use and focused on systems engineers; it also offers integrated optimisation, calibration and results analysis functionalities.

Other specialised platforms also exist in this field, such as AVL CRUISE (oriented to energy efficiency simulation of complete vehicles, including electric vehicles and fuel cells) or tools focused on power electronics such as PLECS or Synopsys Saber, among others. However, in this analysis we will focus on the generalist platforms mentioned above, which are the most widely used for comprehensive physical Modeling of EVs.

2.4.2 Physical Modeling Capabilities and Multi-Domain Approach

A differentiating factor between the tools is their Modeling paradigm: acausal vs. causal, and the intrinsic support of multiple physical domains in a single model.

- **Modelica**: is based on acausal Modeling by equations. The user defines the physical laws (e.g. Kirchhoff equations, conservation of energy, etc.) within components, and the connections represent bidirectional flow/potential exchanges (current-voltage, torque-velocity, temperature-heat flow, etc.). This allows for a natural integration of multiple domains: for example, a battery model in Modelica can simultaneously include the cell's electrical circuit, its heat balance and even degradation kinetics equations, all consistently connected. Modelica is domain agnostic, so electrical, mechanical and fluid coexist without the need for artificial partitions. In addition, the language supports hierarchies

and object-oriented reuse, making it easy to build complex systems from sub-models. In summary, Modelica was designed for general-purpose, multi-domain physical Modeling, with acausal equations offering flexibility to reconfigure models without re-specifying inputs/outputs.

- **MATLAB/Simulink + Simscape:** Simulink itself uses a causal Modeling approach: each block has defined inputs and outputs, which is suitable for representing signal flows in control systems, but less suitable for Modeling physical laws (which are often inherently acausal). The introduction of Simscape added the capability of physical Modeling using acausal networks within the Simulink environment. Simscape provides conserving ports analogous to Modelica connectors: by connecting Simscape components (e.g., a resistor to a battery), the charge and energy conservation equations are automatically established at that node, without the user specifying the direction of flow. Simscape adopts the same principle of implicit equations as Modelica, but encapsulated by domain in different libraries (Electrical, Mechanical, Thermal, etc.) within MATLAB.

  It is important to note that Simulink + Simscape is still less open than Modelica: the acausal equations exist behind the Simscape components, but there is no unified multi-domain language that the user can freely extend beyond using the Simscape Language syntax in MATLAB to create new physics blocks. Still, in terms of Modeling capabilities, Simscape allows combining domains (e.g., an electric motor with electrical circuit and mechanical shaft, coupled to a thermal model), like Modelica, achieving integrated multi-physics models.

- **Simcenter Amesim**: is also a multi-domain platform of acausal nature in the connection of components. Historically, Amesim was based on the formalism of bond graphs and equations assigned to components solved by implicit numerical methods. The user assembles the model with icons of components (pumps, motors, batteries, etc.) connected by ports; each port imposes a stress/flow interaction between connected components. Causality is solved automatically: that is, the software decides internally which variable will be calculated as a function of which, to solve the system. In practice, this gives Amesim similar capabilities to Modelica/Simscape in terms of coupling different domains without manual effort. One difference is that Amesim originally did not expose a Modeling language to the user (everything was done via pre-defined components), although it now allows importing or writing components in Modelica and other languages. Amesim comes with extensive multi-domain libraries, so an engineer can put together, for example, a complete battery-motor-inverter system by selecting library components, without having to handle equations.

- **GT-SUITE**: has multi-domain capabilities but with a slightly different approach due to its legacy in engines. GT uses a simulation engine with different specialised solvers for different sub-domains (fluid, mechanical, electrical) within the same model. For example, it integrates electrical circuit solutions (for cable networks, batteries and inverters), 1D flow solutions for gases and refrigerants, and discrete elements for mechanical kinematics. The construction of the model is by means of 1D diagrams where volumes, ducts, resistors, etc.

are connected. Like Modelica/Amesim, the physical laws (e.g. 1D Navier-Stokes equations in a duct, or RC circuit equations in a battery) are predefined in components, which are connected by establishing flow balances. There is no need to define the direction of the variables: GT determines how to solve the networks. It is therefore acausally multi-domain in practice (although internally it can partition the solution). A strength of GT is its fusion of 1D with 3D: it allows importing CAD geometries or CFD/FEM results to generate equivalent 1D models, being able for example to derive a 1D cooling network from a 3D CAD model of a battery pack. This extends the scope of physical Modeling, combining 1D speed with 3D detail where necessary.

- **ANSYS Twin Builder**: its philosophy is hybrid. On the one hand, it acts as a Modelica environment: it directly supports the inclusion of Modelica models within its schematic (in fact it comes with the Modelica Standard Library and its own Modelica libraries). On the other hand, it retains capabilities of its predecessor Simplorer, allowing SPICE-type circuit diagrams (especially useful for power electronics) and control block diagrams. In Twin Builder one can, for example, connect an electric motor model written in Modelica with a PID controller in block diagram and an IGBT firing circuit in VHDL-AMS, all in a unified simulation. This combination of acausal and causal in the same environment is unique to Twin Builder. As for multi-domain, it is fully capable: electrical, mechanical, hydraulic, thermal - with the advantage that ANSYS provides coupling with its 3D tools (you can include reduce-order sub-models of 3D electromagnetics, structures, CFD, etc., generated in Maxwell, Fluent, etc.). In sum, Twin Builder stands out for its flexibility in Modeling approaches (from Modelica equations to 3D reduced-order models), which gives it strong multi-domain support.

For EV applications, it is common to involve model interaction of multiple subsystems: the battery (electrochemical/electrical + thermal), the power electronics (electrical + digital control + thermal losses), the electric motor (electrical + mechanical + thermal), the transmission (mechanical) and eventually the complete vehicle (vehicle mechanical dynamics with controls) [28], [8], [20]. Tools with solid multi-domain support and acausality allow building integrated models where all these parts coexist in a consistent way.

Modelica provides perhaps the most unified experience in this respect: everything is expressed in a single declarative language, with great freedom to create custom components if they do not exist. Simulink/Simscape achieves something similar but fragmented into different libraries and with some friction if custom components are required (they must be programmed in Simscape language). Amesim and GT-SUITE bring the convenience of very complete libraries, designed specifically for automotive use cases, minimising the need to program equations - at the cost of being more closed environments (although Amesim and Twin Builder partially compensate by accepting Modelica, inheriting some of its openness).

2.4.3 Simulation Precision and Performance

The accuracy of the simulations and numerical robustness (ability to solve complex systems of equations in a stable way) are critical considerations when evaluating these tools. All the environments analysed can, in principle, produce very accurate results if the physical models are well set up and calibrated. Differences arise in the numerical methods available, the ease of handling stiff systems and the default tolerances they employ, which impact on the fidelity and stability of the solutions.

- **Modelica (Dymola/OpenModelica):** Modelica tools typically use advanced solvers of DAEs (e.g. DASSL, IDA) with variable step and event detectors, complemented by symbolic processing that simplifies equations prior to simulation. Dymola performs index reduction and algebraic optimisation techniques that improve stability and speed, reducing cumulative numerical errors. This results in very robust simulations even when dynamics of different time scales coexist (for example, the slow evolution of the temperature of a battery together with fast switching of an inverter). In terms of accuracy, Modelica does not impose simplifications: one can enter very detailed equations (even small ODEs for transient effects within a cell) and the solver will solve them together. Comparative studies have shown that Modelica can match or exceed the accuracy of other tools [29].

- **Simulink/Simscape**: Traditional Simulink uses ODE solvers (Runge-Kutta, BDF, etc.) for causal systems. When Simscape is introduced, implicit equation systems appear that require algebraic solvers in the loop. MathWorks provides specialised solvers for Simscape (e.g. ode15s or other implicit integrators), plus the option of discrete local solvers for certain physical networks. In terms of accuracy, Simscape can achieve very precise solutions, but sometimes needs careful settings: for example, setting very tight tolerances or small maximum integration steps to capture fast transient effects (such as current pulses in an inverter). A common challenge with Simscape is solver tuning for rigid systems: if a battery has time constants of hours in thermal but microseconds in an electronic circuit, the variable solver can face difficulties in efficiently resolving both extremes. MathWorks has introduced improvements (such as automatic stiff mode), but users report that large Simscape models may require manual adjustments for convergence.

  This suggests that, with default parameters, Simscape may have introduced more error or would require a smaller step to match the others. Nevertheless, it is capable of high accuracy if properly configured. On robustness, Simscape includes an initialisation solver to find consistent initial conditions, similar to Modelica. However, users may encounter initialisation error messages or singularities in Simscape if the model is over- or under-determined, problems analogous to those that arise in Modelica (e.g. under-conditioned systems of equations). With good practices (using Simscape's memory blocks or initial conditions appropriately, etc.), Simscape is robust. It should be noted that Simulink (without Simscape), if one models physics manually with blocks, can be more prone to user errors in the equations, affecting accuracy; thus, Simscape is key to robustness in physics models within Simulink.

- **Amesim**: It is renowned for its numerical reliability in industrial contexts. Its components are pre-validated and often include internal algorithms to improve stability (e.g. numerical damping on certain valves to avoid unphysical oscillations). Amesim allows to choose several types of solvers (explicit, implicit, fixed or variable step) depending on the model and objective (fast simulation vs. accuracy). In general, Amesim behaves robustly even with large models, and is tolerant to initial configurations (it has a good steady state solver to start simulations). In accuracy, as we saw, it can replicate experimental data well after calibration. One interesting aspect: being an application-focused tool, it sometimes employs specific numerical tricks - for example, in a combustion engine, it limits steep derivatives - to keep the simulation stable at the cost of more than sufficient accuracy for engineering but without overloading the solver. This contrasts with Modelica, which by default is more purist in solving equations as they are formulated. In short, Amesim offers high practical accuracy and excellent robustness, with a bit of numerical conservatism to avoid problems.

- **GT-SUITE**: Like Amesim, GT prioritizes robustness in industrial scenarios. It can simulate engines in cycles of thousands of combustions without diverging, or complex cooling systems. It achieves this with a combination of specialized solvers: for example, it can use fast explicit integrators for gas flows (where small errors are averaged), and implicit integrators for very stiff loops (e.g., RC electric circuit of a battery). GT allows the user to specify convergence criteria, relaxation, etc. In accuracy, it is highly reliable in its strong domains (e.g. prediction of pressure drops, temperatures, battery SOC) because its models are calibrated with real data frequently. For very fast phenomena (e.g., switching pulses at 20 kHz), GT may prefer that the user use an average model (e.g., an average inverter rather than simulating each pulse) to maintain robustness and not sacrifice time, although in Twin Builder or Simscape one could simulate each pulse with small step. That is, GT tends to balance precision with stability and speed depending on the target. Overall, properly configured, GT-SUITE can be as accurate as the others in most metrics.

- **Twin Builder**: Incorporating Modelica, VHDL-AMS and SPICE, it also inherited their solver capabilities. Simplorer (core of Twin Builder) was known for its prowess in simulating power electronics combined with controls. It offers continuous-discrete integration solvers that handle events (switch on/off) well without losing accuracy. In addition, Twin Builder facilitates co-simulation with ANSYS 3D; for example, you could run a 3D finite element thermal model alongside a 1D battery model. In such cases, the overall accuracy will depend on the synchronization between solvers, but ANSYS provides error-controlled co-simulation methods. In general, Twin Builder can be relied upon for accurate results, backed by ANSYS algorithms (famous in FEM/CFD areas) now applied to 0D/1D. Robustness is also high, although with the caveat that the flexibility of languages (Modelica, etc.) means that the user has more responsibility for ensuring that the model is well formulated.

In conclusion, all platforms can achieve high accuracy if the models are properly constructed. There is no absolute most accurate, since the physics represented is the

same, the difference is in the ease of achieving that accuracy.

2.4.4 Compatibility, Integration and Standards

The capacity of each tool to integrate into wider development flows is crucial. In the automotive domain, a plant model (battery + engine) is rarely isolated: it often interacts with control models (ECUs), with other subsystems (e.g. complete vehicle model) and even with upstream (CAD, detailed circuits) or downstream (HIL, real time) design tools. For each tool, the compatibility with other environments, the support of open standards (especially FMI/FMU for model exchange), and the possibilities for co-simulation and reuse will be evaluated below.

- **Modelica:** Since Modelica itself is an open standard, its philosophy promotes model portability. Using the FMI (Functional Mock-up Interface) standard, virtually all Modelica tools (Dymola, OpenModelica, etc.) can export the model as FMU (Functional Mock-up Unit) for either model exchange or co-simulation. This means that a model created in Modelica can be packaged and then imported into other compatible platforms. For example, it is common to export a Modelica plant model (battery-engine) and import it into Simulink as an FMU block for testing with MATLAB-developed controllers. Modelica, through FMI, manages to integrate with Simulink, LabVIEW, Python, Java, etc., in a standard way. In addition, Modelica offers other ways: tools such as Dymola have APIs to interact with MATLAB, Excel, Python (to run simulations, sweeps, etc.). In co-simulation environments, Modelica can be both master and slave in FMI schemes. This makes it easy, for example, to split a problem: simulate the battery in Dymola and the motor/inverter in another tool, synchronising them by FMI.

  In addition to FMI, Modelica is highly reusable because it is not vendor-bound: the .mo or. mdl models you create can be opened with any Modelica tool (as long as you have the same libraries). This ensures longevity of the models and avoids lock-in. Additionally, Modelica supports interaction with hardware: executables can be generated to run on real-time platforms (dSPACE, xPC Target) and there is even support for Modelica real-time. In terms of control, although Modelica is not a very interactive driver design environment, it does allow drivers to be incorporated (in the form of causal block diagrams within Modelica, or by importing external C/algorithms). For example, a user can implement a PI control in Modelica or import a calibrated control table. However, most prefer to develop the control in MATLAB and use Modelica for the plant - something that, as mentioned, is entirely feasible via FMI.

- **Simulink/Simscape:** Integration is one of its strongest points, especially in the context of control and signal design. Simulink is the industry standard for developing and verifying control algorithms (battery management, motor control, energy strategies, etc.). The advantage here is that the physical model (Simscape) can run in the same simulation environment as the controllers without the need for additional interfaces. For example, an engineer can co-simulate the battery pack in Simscape along with the BMS model in Stateflow, the PWM-controlled inverter in Simulink and a vehicle model in Simulink, all within a single diagram. This native integration reduces complexity and

potential coupling errors. In addition, Simulink supports integrations with CAD tools (via data import, e.g. suspension geometries in Simscape Multibody), with requirements software, and very importantly: with Hardware-in-the-Loop (HIL) environments. With Simulink and Simscape, it is straightforward to generate optimised C code (using Simulink Coder and Simscape Real-Time) to run the model on a HIL platform in real time.

Many companies use this way to virtually test electric vehicle ECUs: the Simscape plant model runs in a real-time simulator while the physical ECU interacts with it. Regarding FMI, MathWorks was initially reluctant to adopt it, but today it offers support: there is FMU Import and FMU Export (as free add-ons from 2019+) that allow Simulink to be used as FMU master or slave. In practice, Simulink can import Modelica FMUs (co-simulation) or export a Simulink subsystem as FMU (usually for co-simulation, as exporting Simscape models for model-exchange has limitations). This has improved compatibility, although it is not as transparent as in Modelica. Another valuable integration is with major system design tools: Simulink can connect with data management software, optimisation (MATLAB environment) and even co-simulate with Simcenter Amesim, GT or others via vendor-provided interfaces (e.g. GT-SUITE offers S-functions for Simulink and Amesim has Simulink Interface blocks).

Thus, Simulink often acts as a hub where different exported models converge. In short, in compatibility Simulink shines with its MATLAB ecosystem (data processing, control design, graphical interface) and supports enough standards (FMI, C code, etc.) to not be isolated.

- **Twin Builder**: It is presented as an open solution in terms of standards: we have already mentioned its support for Modelica and VHDL-AMS, standard Modeling languages. It also natively supports FMI for both importing and exporting models. This means that Twin Builder can be used to orchestrate co-simulations with third-party models. For example, one could import an FMU of a vehicle model in CarMaker and combine it with a battery model in Twin Builder. Or export a complete Twin Builder model (say a digital battery twin) as an FMU for a customer to run on their system. Additionally, Twin Builder integrates tightly with the ANSYS portfolio: it is possible to co-simulate with ANSYS Fluent, Mechanical, Maxwell, etc. using proprietary links. A relevant use case for EVs is to couple Twin Builder (1D system) with a 3D CFD model of battery cooling in Fluent - Twin Builder manages the electrical and basic thermal part, while Fluent calculates detailed temperature distribution, exchanging results at each step.

  This type of multi-scale integration is an added value. Also, using ANSYS SCADE, Twin Builder can incorporate certifiable control logics (e.g. motor control strategies in self-coded SCADE code) and verify the complete system. As for HIL, ANSYS offers outputs to dSPACE or NI platforms, so a Twin Builder model can be prepared to run on real-time. Its adoption of open standards means that we are not forced to use only ANSYS tools; for example, Twin Builder can generate a model in C code for inclusion in another environment or take advantage of Python/Matlab scripts for automations. This

open and integration philosophy is one of the reasons why Twin Builder is called Twin Builder: it is designed to integrate with the real world and with various data sources.

- **GT-SUITE:** Although it is a proprietary environment, it has evolved to coexist with other tools. It is commonly used in conjunction with Simulink: GT allows models to be exported as Simulink S-Functions or even directly as FMUs (currently supports up to FMI 3.0). For example, a complete plant model made in GT (engine + battery + vehicle) can be exported as a co-simulation FMU and run inside a Simulink schematic containing the controllers. Many OEMs use this approach: they design the detailed plant in GT, but test the controls in Simulink where they have their algorithms. On the other hand, GT-SUITE integrates with CAD/CAE software: it can import geometries, CFD results (e.g. radiator flow maps) and also export data for cross validation.

  GT-SUITE has its own co-simulation server called GT-COE, which makes it easy to connect multiple instances of GT with other runtime tools, synchronising them. Regarding open standards, outside of FMI, GT does not expose a general language (its models are stored in proprietary. gts format). However, it offers APIs in Python and MATLAB to handle simulations programmatically, which is useful for parametric or optimisation studies. In addition, GT incorporates model reuse tools such as templates and sub-models that can be shared within a company. In sum, GT makes sure not to isolate the user as some validations are done on third-party platforms, and therefore provides connectors (co-simulation, FMI) to accommodate it.

- **Amesim Simcenter:** Traditionally, Amesim has coexisted closely with MATLAB/Simulink. It provides a module called Simulink Interface that allows an Amesim model to be easily converted into a Simulink block, and vice versa (co-simulate). Before FMI, this was the most common way: for example, an engineer could assemble the EV plant model in Amesim and then co-simulate it with the controller in Simulink via this dedicated link. Today, Amesim fully supports FMI (it was an early adopter in the 1D world). You can export Amesim subsystems as FMUs for external use or import FMUs into an Amesim diagram. In fact, by supporting Modelica internally, importing models is easier (Modelica models can be imported directly). In addition, Amesim has integration with other elements of the Siemens Simcenter portfolio: for example, with Simcenter STAR-CCM+ (CFD) for 1D-3D fluid dynamic co-simulation, or with Simcenter Prescan (autonomous driving simulation environment) to include plant models in traffic environments.

  For the control part, Siemens offers solutions like Simcenter AMESim Control, but the reality is that most users use MATLAB/Simulink for control and Amesim for plant, integrating them via FMI or co-simulation. An interesting point is the customisation via open languages: Amesim allows embedding Python, C or Modelica scripts in components, which means that an expert can add his own algorithm (e.g. an advanced SOC calculation in Python) inside a model and share that component. This flexibility, combined with FMI, makes the reuse of Amesim models quite good within the industrial ecosystem (although not so much academically, as it is an expensive commercial tool).

Regarding HIL, Siemens also provides solutions to run Amesim models on real-time platforms, or via C code export, so that it is not just a desktop simulation.

In a nutshell, **all the tools analysed allow integration with other tools, but Modelica and Simulink stand out by nature**: Modelica, for embracing open standards such as FMI and being multi-tool, and Simulink for being the de facto standard for integration with control and having a multitude of bridges with other applications. Amesim, GT and Twin Builder have followed the trend by opening to FMI and Modelica, which levels the field quite a bit in terms of compatibility. An engineer can mix these tools in one process (e.g. use Modelica for detailed Modeling of certain subsystems and GT for others, coupling them via FMI). Model reuse is easier and cleaner in Modelica, thanks to object orientation (e.g. inheriting from a base electric motor model to create variants). In Simulink/Simscape it is possible to reuse subsystems and masks, but without a mechanism as powerful as Modelica's inheritance. Amesim allows reuse of submodules but within its environment, similar GT with parametric templates. Twin Builder, by using Modelica, inherits Modelica code reuse. Thus, Modelica leads in open standardisation and reuse, with Twin Builder leveraging those same standards, while Simulink/Simscape leads in integration with the control and system design stage.

2.4.5 Computational Performance and Scalability

Finally, let us compare computational performance, i.e. simulation speed and the ability to scale to large models (many components or long simulations), including real-time simulation possibilities. In the aforementioned comparative study, concrete performance data were obtained: using the same battery aging model, running 10 years of simulated operation, Dymola (Modelica) was the fastest by far, followed by Simulink, next Amesim and the slowest was Simscape [29].

In numbers: Dymola completed the 10-year simulation in 288 seconds, while Simulink took ~3-5 times as long, and Simscape even longer. This is evidence that the Modelica implementation (Dymola) took advantage of its optimisation to efficiently solve even a long time horizon. Simscape, being heavier on implicit equations, had difficulties to match that speed.

- **Modelica/Dymola**: Its performance strengths come from symbolic processing and highly optimised code generation in C. Prior to simulation, Dymola simplifies the system of equations, reduces indices and eliminates unnecessary algebraic states. This results in models that are more compact and faster to solve. In addition, Dymola supports parallelisation of certain parts (e.g., when compiling, it can parallelise loops on multiple cores if it detects independence). In terms of scalability, it is known that Modelica can handle models with tens of thousands of equations. For example, simulating a battery pack with 100 individual cells is feasible in Modelica; although it will take longer than representing the pack with 1 equivalent cell, the tool will be able to solve it. In fact, one of the benefits of Modelica is that it allows you to vectorise components.

  For example, 100 identical cells can be modelled in series using an array of components, which the compiler can exploit to generate efficient loops in the resulting C code. There are reported cases of Modelica running power grid

models with tens of thousands of nodes in acceptable time, thanks to its robustness on large systems. For real-time simulation, Dymola has an optimised code generation mode (using fixed-step solvers, eliminating dynamic allocation, etc.) and has been used in HIL simulations with success. Obviously, achieving HIL with a detailed full battery-engine model may require simplifications (e.g., using a simpler solver, or reducing the size of the model), but it is possible. Modelica's scalability is also shown in parametric or Monte Carlo sweeps [9]: its Python API allows launching multiple simulations in parallel with different parameters [30], taking advantage of multi-core hardware or clusters - very useful when exploring manufacturing variability in batteries, for instance.

- **Simulink/Simscape:** Simulink as a platform is very efficient for causal control models (which tend to be computationally light). When Simscape is introduced, the computational burden goes up because of the implicit systems to be solved. Simscape has improved over the years in solver, but there is still some penalty. For example, doubling the number of Simscape components does not always scale linearly in time, sometimes worse because of the increasing difficulty of solving the system matrix. In moderate models (tens of components), Simscape works well; in large models (hundreds of explicit cells, for example), it can become slow or even unstable. MathWorks recommends in such cases using sub Modeling techniques (e.g., grouping cells into submodules and reducing thermal nodes) to keep the simulation manageable. One area where Simulink is excellent is in co-simulation and real-time: it integrates with simulation hardware easily, and one can migrate parts of the model to FPGA or similar if required. However, when complexity is high, one sometimes opts for decoupling: e.g., simulating the battery on a separate microcontroller from the motor, etc.., in HIL. Simulink supports this with its Simulink Real-Time tool. In general, Simulink scales well at the complete system level (that is why it is used for virtual vehicle integration), but the physical part is its limit: you do not usually simulate every detail in Simulink because it can become slow.

  For intensive calculations (e.g. magnetic motor saturation, or conjugating hundreds of cell ODEs), Modelica or manual C++ sometimes perform better. One indicator: in the benchmark, Simulink was 3 times slower than Modelica in simple scenarios, and 5 times slower in long scenarios [29]. Still, it should be noted: Simulink can benefit from tricks like compiling to native code (accelerator mode), using fixed solvers with steps calculated for the case, and with that you can get closer. In HIL, many companies use simplified Simscape models (e.g., reduce a 100-cell pack to 10 representative cells) to meet the computation cycle in 1ms. Simscape has a configurable Local Solver in some subsystems to parallelise (e.g., simulate the battery pack with a separate solver in parallel to the rest of the model), which can take advantage of multi-core. These are advanced solutions, but they demonstrate that Simulink/Simscape can be adapted to scale reasonably, albeit with more manual intervention than Modelica.

- **Amesim**: In performance, it ranked between Simulink and Simscape in the above-mentioned study. Its simulation engine is efficient for many classes of problems but can be challenged by large systems. Amesim is optimised in components of its libraries - many use simplified numerical methods that allow

for larger time steps without losing stability. For example, in hydraulics, Amesim can use specific implicit methods that allow 1e-3s time steps where Modelica might need 1e-4s. In electrical, if not simulating switching of each transistor, Amesim can solve an inverter as an average element and move forward with large steps. Therefore, Amesim scales well in full system models and can often run faster than Simscape equivalents (as seen). Where it might have limitations is in real time: although Amesim models have been used in HIL, co-simulation with Simulink is sometimes used for that task or C code export.

Siemens offers a FMU tool for HIL that takes Amesim models and prepares them for real time [31]. Regarding parallelisation, there is not much public information; it is known that some Amesim solvers can use multi-threading internally, but in general, scalability is achieved more by numerical robustness than by parallelism. In any case, Amesim has proven to be able to simulate large models (complete vehicles with multiple subsystems) with acceptable performance, especially when the model is fit for purpose (e.g., not overloaded with unnecessary detail in each part).

- **GT-SUITE**: GT is designed for efficiency in the simulation of complex systems since its origin is to simulate engines in real time. It uses several tricks: it linearizes certain parts, uses fast explicit integrators when it can, and allows macro-stepping (large jumps) in slow submodels while computing fast ones with small steps. The user has control to define priorities (e.g., simulate combustion dynamics with 0.1° crankshaft step but the rest with 1° step). In an EV, you can simulate the electronics with a finer pitch and the thermal with a thicker one, all in the same run. GT also supports distributed computing: you can distribute subsystems on different threads or even machines (which is used for co-simulation with third parties but could also be used to parallelize within GT certain decoupled loops). As for HIL, Gamma Tech has GT-RealTime, which is a guide for exporting optimized models to dSPACE platforms, etc. Many manufacturers run GT models of engines on HIL to test engine ECUs. For EV, this extends to testing the inverter ECU with a GT model of the engine and vehicle.

  Typically, some parameters must be adjusted (e.g. setting explicit integrators, pre-interpolated tables, etc.), but GT can achieve run times on the order of microseconds per step for medium-sized models. On scalability, GT can handle integrated models of the whole vehicle: there are examples where they simulate engine, battery, transmission, cabin together. Its hierarchical solver handles the load well. GT is not necessarily always the fastest, for purely algebraic-differential equations as in Modelica, Dymola can optimize more globally. But for certain automotive applications, GT is tuned to be efficient. For example, to iterate engine designs on a WLTP cycle repeatedly, GT is often the tool of choice for its speed in those specific calculations (and its integration of parametric optimization). In EV, we might see something similar: to optimize a battery design and its cooling through 100 drive cycle simulations, GT can handle it with its batch execution engine and robustness.

- **Twin Builder**: On performance, there is no public comparative data, but we can extrapolate. Twin Builder by supporting Modelica inherits in part its efficiency, but its flexibility (multi-solver, co-sims) can add overhead. ANSYS has surely worked on optimizing it for digital twins running in the cloud in near real time [32], [33]. Its ability to generate (reduced) ROM models to use instead of complex 3D models is a scalability strategy: better to spend time pre-computing a ROM than then running a slow model repeatedly. Twin Builder also has sensitivity analysis and DOE tools that take advantage of multiprocessing. In the end, it is reasonable to assume that Twin Builder can simulate an EV powertrain in near-real time, given that it is marketed for operational twins. And with its control software integration, they expect it to run fast enough to interface with controllers. However, compared to pure Modelica, Twin Builder is likely to introduce some overhead, depending on how optimized its libraries and solver integrator are.

2.5 Modelica and its Alternatives

After this comparative analysis, we can conclude that each tool has particular strengths, but Modelica (as a language supported by environments such as Dymola) offers several unique strengths that give it notable competitive advantages in the development of battery and electric powertrain models for vehicles:

- **Unified multi-domain model**: Modelica allows the entire EV system to be described with a single declarative language, natively integrating electrical, mechanical, thermal and control. It does not require partitioning the problem or using multiple tools - a single Modelica model can include everything from simplified cell electrochemistry to vehicle dynamics to power electronics. This integrated approach reduces interface errors and ensures physical consistency in the interactions between subsystems. Other tools also achieve multi-domain integration but often require external couplings (Simulink with different toolboxes, co-simulation, etc.) or are focused on certain domains rather than others. Modelica by design is generic and extensible, which is ideal in a field such as electromobility where several engineering fields are involved.

- **Acausality and model reuse**: Modelica's acausal approach makes it quite easy to reuse models in different configurations. For example, the same battery module submodel can be reused in different pack sizes by simply connecting more modules in series/parallel, without modifying internal equations. If tomorrow the topology of the system changes (e.g., a second electric motor is added to make an AWD), in Modelica it is a matter of connecting that new motor to the rest; the solver will redistribute the equations without the modeler having to rework inputs and outputs. This flexibility is not so easy in causal environments (where adding a component often means redesigning the signal scheme). Modelica also supports the creation of modular libraries: companies can develop their own component libraries (for example, a proprietary battery cell model) and reuse it in all their projects, and even update it in one place so that all models using it benefit from the improvement. This modularity is an important asset in long-term or variant projects (think of vehicle platforms sharing engine/battery with different calibrations; Modelica makes it possible to have a single base model and specialize it with inheritance for each case).

- **Accuracy and fidelity with efficiency:** Modelica has proven to be able to achieve high levels of fidelity without sacrificing performance. The ability to include complex equations (e.g. non-linear temperature dependencies, additional differential equations for degradation phenomena) within the model, and still simulate quickly thanks to symbolic optimization, is a great advantage. In the EV domain, where you want to evaluate long scenarios (battery charge cycles, 10+ year lifetime, etc.), Modelica's efficiency translates into less computational time to obtain results, which speeds up the design iteration. Quantitative studies confirmed that Dymola (Modelica) can be several times faster than the alternatives in simulating long cycle times without loss of accuracy.

  In addition, the robustness of its DAEs solvers allows models with very different dynamics (such as those present in an EV: electronics in microseconds vs. degradation in hours) to be solved together in a stable way, avoiding the need to separate models.

- **Open standards support:** As Modelica is a free standard, its use avoids lock-in to a single tool or vendor. This is strategic for automotive companies looking for longevity of their model investments and compatibility over the years. They can develop a model today in Modelica with Dymola and later decide to run it in another Modelica tool or integrate it into a different digital twin platform, with minimal effort. FMI's native support also makes Modelica work well with almost any workflow: it is easy to hand over an encapsulated Modelica plant model to a control team, or to import a given external vendor's model into Modelica as an FMU. In the EV environment, where many vendors coexist (each might provide a model of their component), this interoperability is crucial. In addition, Modelica's open ecosystem promotes academic collaboration and innovation: many universities research batteries and electric propulsion using Modelica and share their findings in the form of models or publications, thus feeding the state of the art available to the industry.

- **Customisability and state-of-the-art:** Related to the above, Modelica allows new knowledge or effects to be easily incorporated into the model as they become better understood. For example, if a company develops a new algorithm for calculating battery health based on, say, entropy counting, it can implement it in Modelica within the existing model, without waiting for a software vendor to include that functionality. This makes Modelica very suitable for upfront research and development, which is often the source of competitive advantage in electromobility (batteries with better management, motors with finer controls, etc.). Other tools, being more closed, limit the user to what the supplier offers (although they can be extended, it is not usually with the same freedom). In this sense, Modelica future-proofs Modeling efforts: any new physics or component can be integrated by describing its equations.

- **Integration with FMI**: While Simulink leads in control design, Modelica complements that strength rather than opposing it. An optimal flow that many follow is plant model in Modelica (accurate and multi-physics), exported as FMU, and integrated into Simulink to design and test controllers. Thanks to the efficiency of the Dymola FMU, it can be simulated in near real-time within

Simulink [34]. This combines the 'best of both worlds': the quality of the Modelica physical model with the familiarity and power of MATLAB/Simulink for control. In our times where multi-tool collaboration is commonplace, Modelica fits very well. On the other hand, if desired, Modelica also allows controls to be incorporated into the same model (e.g. using Modelica control libraries or importing external logic), offering the possibility of having a self-contained plant + control model ready to, for example, be run as a digital twin in a connected vehicle.

- **Real-time execution and HIL**: Although not unique, Modelica (especially Dymola) has demonstrated that it can compile complex models to efficient C code suitable for running in real-time on hardware simulators. This is essential for HIL testing of inverters, BMS, etc. With the right optimisations, a Modelica powertrain model on a dSPACE platform [35], taking advantage of its superior performance. This coupled with FMI means that even in heterogeneous HIL benches, a Modelica model can be integrated as a component.

However, it is fair to recognise that Modelica also has challenges: the need for qualified staff to master the language and the management of the libraries, as well as the investment in tools (Dymola is commercial, although there are free alternatives with fewer features). However, once the learning curve is overcome, the benefits in flexibility, speed and fidelity tend to justify their use, especially in projects where many simulations will be performed (design optimisation, use cycles, etc.) or where the model will be continuously refined.

2.6 Conclusions

The Modelica alternative tools have significant merits: Simulink/Simscape is almost irreplaceable in loop with control and for rapid industrial adoption, Amesim offers immediate productivity with validated models, GT-SUITE is unbeatable in integrated thermal simulation, Twin Builder facilitates digital twins connected with ANSYS corporate tools. In fact, many organisations use combinations of these tools, assigning each to the role where it is most relevant.

However, Modelica is positioned as a powerful solution for electric vehicle simulation, offering a hard-to-achieve combination of fidelity, flexibility and speed. For organisations willing to invest in initial training and adopt open standards, Modelica can pay big dividends: more comprehensive and adaptable models, reduced computational time, and freedom to innovate. For these reasons, Modelica has been gaining a place in modern automotive, complementing and sometimes replacing traditional tools, especially as simulation challenges become more interdisciplinary (such as marrying battery chemistry with vehicle dynamics and control software). For battery and electric powertrain applications, Modelica competitively offers the ability to cover the entire system with high accuracy and reusability, positioning itself as a key part of the electric vehicle engineering toolset.

A comparative summary of the strengths and weaknesses of the aforementioned tools is shown in Table 2.2.

Table 2.2: Summary Comparison of Modelica and Main Alternative Tools.

Tool	Modeling Approach	Integration and Standards	Performance
Modelica (Dymola, OpenModelica)	Multi-domain acausal equations; Modelica open language.	Open standard; FMI support (model and co-sim), portable models between different tools.	**Very high**: DAE solvers with symbolic optimisation (up to ~5× faster than Simulink in a battery test) [29].
Simulink/ Simscape	Causal block diagram + acausal physical networks (Simscape); separate multi-domain libraries.	Native integration with drivers (MATLAB/Simulink); FMI export (co-sim) available.	**Good with simple models**; can be slower with complex physical models [29].
ANSYS Twin Builder	Hybrid Modeling: supports Modelica (acausal), VHDL-AMS, SPICE and causal blocks in schemas.	FMI support; co-simulation with Ansys 3D tools (Fluent, Mechanical via ROM); integration with SCADE (control software).	**High**: digital twin oriented, including 3D model reduction; robust multi-domain solvers (incl. fast electrical events).
GT-SUITE	Specialised multi-physics 1D simulation; acausal approach with dedicated domain solver.	Import/export models via FMI (co-sim); integration with Simulink (S-function) for control loops.	**Very high in its domain**: optimised for large systems (used in real-time and HIL); proven scalability in models of hundreds of components.
Simcenter Amesim	0D/1D acausal Modeling; connection of components by ports (causality automatically assigned).	Supports FMI; allows C, Python or Modelica models to be included within components; co-simulation with Simulink or other common industry environments.	**High**: Reliable simulation; in comparative studies its performance was intermediate (faster than Simscape but beaten by Dymola in speed) [29].

3 Modeling of the Electric Powertrain

In this chapter, the physical architecture of the system implemented using the *EPowertrain* library is described, detailing the main models developed to represent the energy behaviour of an electric vehicle. The approach adopted seeks a balance between physical realism and computational efficiency, allowing energy consumption to be accurately simulated without incurring unnecessary complexity for the analysis of driving cycles.

3.1 Introduction

In this chapter the mathematical Modeling of the physical systems corresponding to the main components of the *EPowertrain* library is presented. The mathematical Modeling of each of these components is discussed.

The main objective of this chapter is not to describe their implementation in the Modelica environment, which is discussed in Chapter 4, but to provide a conceptual and mathematical basis that justifies the choice and structure of the models used. The aim is to provide the physical and mathematical foundations for their subsequent implementation, which will be described in Chapter 4.

3.2 System Overview

The modelled system represents the powertrain of an electric passenger vehicle powered by a lithium-ion battery. Its objective is to transform the energy stored in the battery into useful wheel motion, through a chain composed of electrical, electromechanical and mechanical components. The overall architecture includes the following main subsystems:

- **Battery:** source of electrical energy.
- **Power converter:** regulates the voltage delivered to the motor.
- **Direct current (DC) motor**: converts electrical energy into mechanical torque.
- **Frame model:** represents the vehicle's body frame inertia and resistance to movement.

The interaction between these elements is based on the conservation of energy and the continuity of physical variables such as voltage, current, torque and angular velocity.

In an electric powertrain, the energy flow analysis is essential to evaluate the overall efficiency, to identify the main sources of losses and to study phenomena such as energy recovery through regenerative braking. An adequate representation of these flows allows for more realistic simulations and optimisation of the design and energy management of the electric vehicle (EV).

The Modeling of the energy flows in the powertrain developed considers the main stages of energy conversion and transfer, from electrochemical storage in the battery to its transformation into useful mechanical energy in the wheels, passing through the electronic conversion and control devices. Along this chain, various losses are introduced that affect the overall system performance and must be carefully considered in the model.

The simulated model represents a passenger car-type electric vehicle in a single-drive configuration. The energy chain starts at the battery, which supplies electrical energy through a power converter to a direct current (DC) motor. This motor generates torque, which is applied to the chassis, represented by a simplified kinematic model.

The main energy flows represented in the *EPowertrain* library are as follows:

- **Battery discharge flow** corresponds to the transfer of positive current from the battery to the power converter, and then to the electric motor to generate motion. This flow implies a decrease in the state of charge (SoC) of the battery.

- **Regenerative braking charge flow**: during deceleration phases, the motor can act as a generator, reversing the direction of current flow and allowing some of the kinetic energy to be recovered as a battery charge.

- **Resistive losses**: modelled as voltage drops associated with the internal resistance of the battery and power electronics such as converters and inverters. These losses manifest themselves as heat generation not explicitly modelled in this version of the library.

- **Switching losses**: in the present model, the fast-switching effects of semiconductor devices have been idealised through continuous behaviour, eliminating discrete state change events and favouring the numerical efficiency of the simulations.

- **Mechanical losses**: represented by viscous friction and dynamic inertia of the drive system, they affect the conversion of electrical energy into usable mechanical energy.

The energy balance of the system can be expressed in simplified form by the following relationship:

$$P_{bat} = P_{motor} + P_{looses} \tag{3.1}$$

where P_{bat} represents the net power supplied to or absorbed by the battery, P_{motor} is the useful power converted into mechanical work, and P_{losses} groups the different sources of inefficiency present in the system. It should be noted that, to maintain the focus on energy efficient simulation, the following simplifications have been adopted:

- Neither the detailed thermal behaviour of components nor their effect on electrical or mechanical losses has been modelled.

- Switching losses of power electronics have been represented continuously, without introducing discrete events associated with PWM operation of inverters or converters.

- The thermal dissipation of the electric motor and the influence of temperature on its behaviour have not been explicitly considered.

These Modeling decisions make it possible to accurately simulate energy consumption under complete driving cycles, optimising simulation times without sacrificing the validity of the results from an energy point of view.

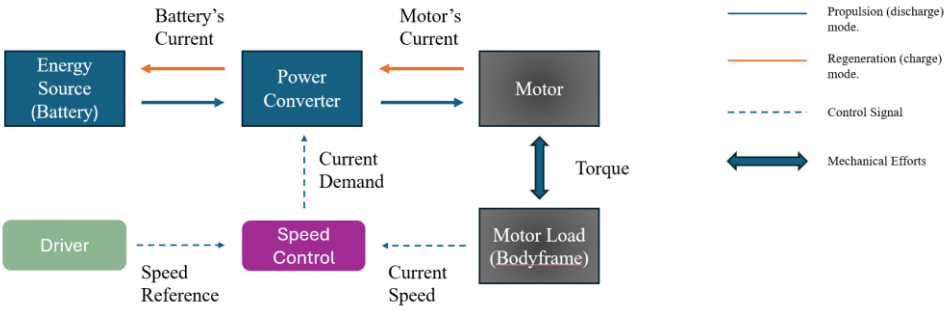

Figure 3.1: Typical EV Power Flows.

3.3 DC Motor

The Motor model corresponds to the dynamic behaviour of a permanent magnet direct current (PMDC) motor. The main phenomena represented in the model are:

- Armature voltage drops due to resistance and inductance.

- Generation of counter-electromotive force proportional to the angular velocity of the rotor.

- Torque production proportional to the current in the winding.

- Effects of viscous friction and inertia dynamics of the motor shaft.

The motor is represented by the following coupled equations, which represent the electromechanical interactions corresponding to a DC motor:

$$V_m = R_m \cdot I_m + L_m \cdot \frac{dI_m}{dt} + K_e \cdot \omega \tag{3.2}$$

$$J \cdot \frac{d\omega}{dt} = K_t \cdot I_m - T_{load} - B \cdot \omega \tag{3.3}$$

Where corresponding parameters are shown on table 3.1.

Table 3.1: DC Motor Parameters.

Parameter	Symbol	Unit
Motor's applied voltage	V_m	V
Motor's current	I_m	A
Winding resistance	R_m	Ω
Winding inductance	L_m	H
Counter-electromotive constant	K_e	$V \cdot s/rad$
Shaft's angular speed	ω	rad/s
Rotor's inertia	J	Kg/m^2
Torque constant	K_t	$N \cdot m/A$
Shaft's load resistance	T_{load}	$N \cdot m$
Viscous friction coefficient	B	$N \cdot m \cdot s/rad$

3.4 Voltage Regulator

In power converter Modeling, it is common to represent output voltage control using pulse width modulation (PWM) strategies. However, this approach involves introducing many discrete switches in the system, generating discontinuities in the state variables and in the current and voltage flows.

These discontinuities significantly increase the computational load and simulation time for several reasons:

- They force the numerical solver to detect and handle high frequency events.
- The need to reduce integration steps to correctly capture the jumps.
- Can make convergence and numerical stability difficult in complex multi-domain models.

Since the purpose of this work is focused on energy analysis and not on the detailed representation of electronic switching, we have chosen to idealise the behaviour of the converter. The approach adopted is to model the converter as an ideal voltage adaptor, where the output:

$$V_{Out} = D \cdot V_{In} \tag{3.4}$$

This idealization presents some benefits:

- Discrete event generation is eliminated, favouring continuous dynamics.
- Simulation efficiency is significantly improved, allowing complete standardised driving cycles (WLTP, UDDS) to be performed with reasonable times.
- It facilitates the reconfiguration and extension of the model for future simulation scenarios.

In addition, an energy balance equation has been introduced to ensure that the conservation of energy in this model is fulfilled.

$$\frac{dE_{Balance}}{dt} = P_{In} + P_{Out} = V_{In} \cdot I_{In} + V_{Out} \cdot I_{Out} \tag{3.5}$$

Table 3.2: Voltage Regulator Parameters.

Parameter	Symbol	Unit
Output voltage	V_{Out}	V
Input voltage	V_{In}	V
Output current	I_{Out}	A
Input current	I_{In}	A
Energy balance	$E_{Balance}$	J
Modulation factor	D	-
Input power	P_{In}	W
Output power	P_{Out}	W

3.5 Battery Model

The battery model represents an idealised cell with a second-order equivalent circuit structure. This configuration seeks to capture both the static response and certain transient dynamics of the electrical storage system, while maintaining a low computational load.

The equivalent circuit includes:

- A state-of-charge (SOC) controlled voltage source, whose value ranges from a minimum discharge voltage (V_d) to a maximum charge voltage (V_f).
- A series resistor (R_s) that models the internal resistive losses.
- A parallel network consisting of a resistor (R_p) and a capacitor (C), which captures the polarisation effects and dynamic response of the cell.
- An internal calculation of the SOC from the net current supplied or absorbed, considering a total capacity (Cap) and a configurable initial state (SOC_{init}).

Model equations
The main equations of the model are:

$$SOC(t) = SOC_{init} - \frac{100}{Q} \int_0^t I(\tau) \cdot d\tau \tag{3.6}$$

$$V_{oc}(SOC) = V_d + (V_f - V_d) \cdot SOC \tag{3.7}$$

$$V_{Batt} = V_{OC} - I \cdot R_s - V_{RC} \tag{3.8}$$

Where V_{RC} is the voltage drop in the parallel RC network:

$$C \cdot \frac{dV_{RC}}{dt} = \frac{V_{OC} - V_{RC}}{R_p} - I \tag{3.9}$$

Table 3.3: Battery Model Parameters.

Parameter	Symbol	Unit
Open circuit voltage	V_{oc}	V
Series resistance	R_s	Ω
Parallel resistance	R_p	Ω
Capacitor capacitance	C	F
Battery's charge capacity	Q	$A \cdot h$
State of charge	SOC	%
Initial state of charge	SOC_{init}	%
RC-branch voltage drop	V_{RC}	V
Battery cell voltage	V_{Batt}	V
Full-state voltage	V_f	V
Depleted-stated voltage	V_d	V
Delivered current	I	A

3.6 Body Frame Model

The frame model implements a simplified representation of a vehicle chassis, focusing exclusively on its longitudinal dynamics. This simplification is suitable for the purposes of this work, as it allows capturing the main effects of mass, inertia and terrain slope on energy consumption, without introducing the complexity associated with more comprehensive vehicle dynamics models. The model considers the following effects:

- **Longitudinal inertia**: This represents the resistance of the vehicle to changes in its linear velocity, modelled through the equivalent mass m.
- **Resistive forces**: Rolling forces, aerodynamic drag (if extension is desired) and gravitational effects associated with terrain inclination are included.
- **Terrain slope**: A slope angle α is introduced which modifies the component of the gravitational force acting on the vehicle.

The used equation of movement is:

$$F_m = F_I + F_d + F_r + F_g \tag{3.10}$$

Where F_m, F_I, F_d, F_r and F_g represent the forces provided by the engine, the inertia of the vehicle mass, the aerodynamic loads, the friction losses of the tyres and the effects of weight as a function of terrain gradient respectively. The force coming from the

engine can be calculated from the torque applied to the wheels.

$$F_m = \frac{T_m}{R_{wheel}} \tag{3.11}$$

Where T_m is the torque applied by the motor shaft and R_{wheel} is the radius of the wheel. The inertia forces to which the chassis of the vehicle is subjected correspond directly to its speed variation.

$$F_I = m \cdot \frac{dv}{dt} \tag{3.12}$$

Where v is the vehicle speed and m is the vehicle's body mass. The aerodynamic drag force loads are expressed as:

$$F_d = \frac{1}{2} \cdot \frac{C_d \cdot \rho \cdot v^2}{A_f} \tag{3.13}$$

With C_d as the characteristic vehicle drag coefficient, ρ the fluid (air) density and A_f as the vehicle front area. The tyres friction force is estimated as a function of a coefficient and the projection of the vehicle's weight on the ground.

$$F_r = C_r \cdot m \cdot g \cdot cos(\alpha) \tag{3.14}$$

With C_r as the rolling coefficient of the vehicle's tyres. Finally, the contribution of the weight obeys the equation of the inclined plane.

$$F_g = m \cdot g \cdot sen(\alpha) \tag{3.15}$$

In table 3.4, the parameters of the body frame are presented as follows.

Table 3.4: Body Frame Model Parameters.

Parameter	Symbol	Unit
Motor's applied force	F_m	N
Motor's applied torque	T_m	$N \cdot m$
Wheels' radius	R_{wheel}	m
Inertia force	F_I	N
Vehicle's mass	m	Kg
Vehicle's velocity	v	m/s
Drag force	F_d	N
Drag coefficient	C_d	-
Fluid density	ρ	Kg/m^3
Front area	A_f	m^2
Rolling friction force	F_r	N
Rolling coefficient	C_r	-
Weight force	F_g	N
Gravity constant	g	m/s^2
Terrain's slope	α	rad

3.7 Conclusions

This chapter has approached the mathematical formulation of the most important physical systems in the electric powertrain of a passenger car. By means of a functional decomposition in its main components, the fundamental models that allow to represent the energy flow from the electrochemical storage to the mechanical movement in the wheels have been defined.

For each component, the Modeling hypotheses, the differential or algebraic equations that describe its behaviour, and the key variables with their respective units have been established. This description allows capturing the essential interactions of the system in a physically coherent way, facilitating its subsequent implementation in an acausal simulation environment such as Modelica.

Chapter 5 will show that these models, despite their simplicity, offer an adequate balance between physical fidelity and computational efficiency, suitable for system-level simulations and energy consumption analysis.

The simplification of certain effects, such as thermal losses or battery aging, has been a conscious decision to focus on electrical behaviour and energy validation under driving profiles. This conceptual and mathematical framework forms the basis on which the modular implementation of the EPowertrain library, described in chapter 4 below, is structured.

4 The EPowertrain Modelica Library

4.1 Introduction

The *EPowertrain* library has been developed as a central part of this work with the aim of providing a modular, extensible and physically coherent simulation tool for the energy study of battery electric vehicles. Based on the Modelica language and following object-oriented design principles, this library allows to easily compose realistic electric powertrain architectures, with an intermediate level of complexity that balances accuracy and computational efficiency.

The main motivation behind the design of this library is to reduce the limitations found in existing libraries, which are either too general (MSL) or too component-oriented without a common framework for system-level energy simulation. In contrast, *EPowertrain* structures its models by functionality, which allows to compose complete electric vehicles from reusable blocks that can be coupled together in a natural way by means of physical connectors.

This chapter describes the structure of the library, the functional content of each of its packages and the key models implemented.

4.2 Library Structure

The *EPowertrain* library has been designed following a modular and functional architecture, which allows to build, analyse and extend electrical kinematic chains in a flexible and reusable way. Its internal organisation responds to the principle of separation of responsibilities, grouping the models according to their physical domain or logical function within the system. This arrangement favours structural clarity, facilitates error diagnosis and allows individual validation of the different subsystems.

Although the entire implementation is contained in a single Modelica main package, internally the library is divided into well-defined functional groups, which encapsulate components of a similar nature - such as power elements, mechanical structures, sensors or control blocks - and expose coherent interfaces to ensure interoperability. The architecture follows the principles of object-oriented Modeling:

- **Encapsulation:** each component hides its internal equations and exposes only the necessary variables using standard connectors.

- **Inheritance and reusability:** new specialised versions of components can be derived from base classes without modifying the original code.

- **Acausality:** connections between components are made using physical variables (such as voltage-current or torque-speed), without the need to define an explicit direction of signal flow, making it easy to reconfigure the model.

In addition to the main blocks, the library incorporates auxiliary modules for signal routing and data acquisition, as well as a set of example models. These examples illustrate the use of the library in practical cases, such as the analysis of energy consumption under standardised or real driving profiles. This structure responds to the following objectives:

- To allow modular expansion of the library and independent validation of its components.
- Facilitate the configuration of different electric vehicle architectures (varying battery, motor, converters or control strategies).
- Effortlessly integrate the control logic in a multi-domain physical context.

The main subpackages that make up the library, which will be described in detail below, are described below:

- **Interfaces:** Defines the physical connectors used by electrical, mechanical and control components. Ensures physical consistency in the connection of heterogeneous models.
- **SignalRouting:** Contains auxiliary blocks for signal routing and manipulation, such as adders, gains or input generators.
- **Sources:** Provides models of electrical sources and input generators, both ideal and controllable, used to simulate external conditions or experiments.
- **Electrical:** Includes models of major electrical components, such as the battery, converter and passive elements. Represents the electrical domain of the powertrain.
- **Mechanical:** Groups models of the mechanical domain, such as bodies with inertia, ideal wheels and chassis models. Represents the physical interaction with the environment.
- **Control:** Provides functional control blocks, including speed controllers, modulators and table-based reference systems.
- **Sensors:** Provides sensor models for reading physical variables such as current, voltage, speed, torque and energy. They are essential for performance analysis.
- **Examples:** Contains complete example models for validation and demonstration of use. Their detailed analysis is covered in Chapter 5.

The concrete structure of the library and the details of each functional module are described in more detail in Appendix A, where representative fragments of the implemented code are presented and the function of each package is explained.

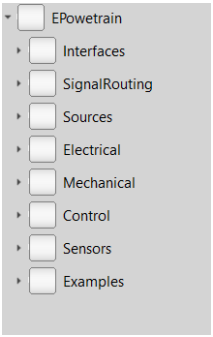

Figure 4.1: EPowertrain Library Main Structure.

4.3 Interfaces

Figure 4.2 shows the Interfaces package, witch contains the physical connectors that allow interactions between models from different domains. The following types are defined:

- Electrical connectors whose variables are voltage and current.
- Mechanical connectors with torque and position variables.
- Generalist input, output or input and output connectors. Intended for the manipulation of real or Boolean signals.

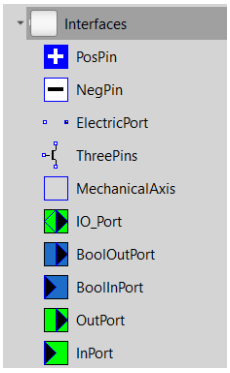

Figure 4.2: Interfaces Package Composition.

4.4 SignalRouting

The SignalRouting package provides signal processing blocks, especially useful for experiments like saturation negation or delays among others. It also includes blocks for multiplexing signals and creating electrical buses. Facilitating the routing of signals in a clean and organised way. These blocks do not model physical phenomena directly but are essential in the modular structure of complete models. The SignalRouting package structure is shown below in Figure 4.3.

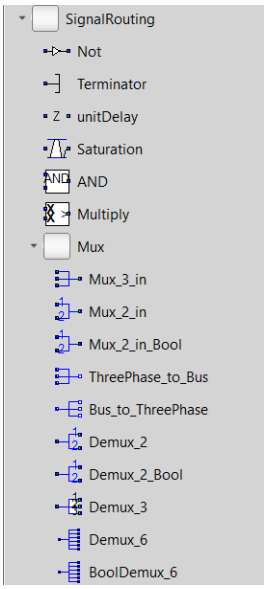

Figure 4.3: SignalRouting Package Structure.

4.5 Sources

The Sources package (Figure 4.4) contains models of controlled or ideal sources, necessary for experimentation and simulation of external conditions as well as for the generation of input profiles.

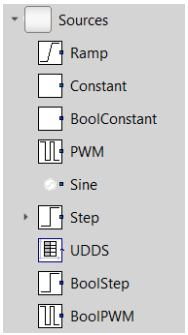

Figure 4.4: Sources Package Models.

4.6 Electrical

The Electrical subpackage is one of the core components of the *EPowertrain* library, as it houses the components responsible for Modeling the electrical behaviour of the propulsion system. These models simulate the conversion, distribution and dissipation of electrical energy in the vehicle's powertrain, from the battery to the engine.

The design of the models contained in this package follows a balanced philosophy between physical realism and computational efficiency. Priority is given to the representation of phenomena relevant to energy analysis (such as voltage drop under load or dynamic motor response), while avoiding the use of excessively detailed models that would introduce discontinuities or unnecessary numerical rigidity, especially in long duration simulations.

All electrical components implement electrical pin connectors, defined in the Interfaces subpackage, which guarantees automatic current conservation and electrical potential continuity for all connections. The main models included in this package are described in Figure 4.5.

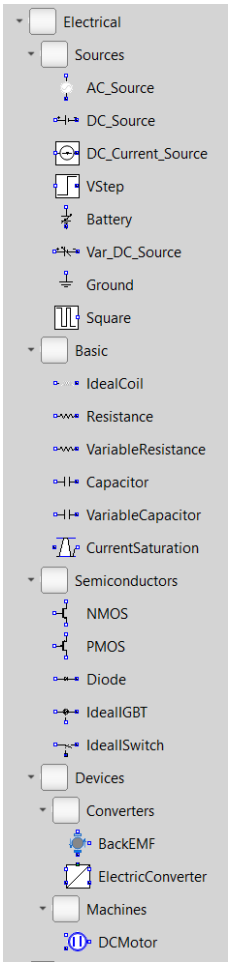

Figure 4.5: Electrical Package Components.

4.6.1 Battery

The battery model implemented in *Electrical.Sources.Battery* represents an idealised cell with a second-order equivalent circuit structure. This configuration seeks to capture both the static response and certain transient dynamics of the electrical storage system, while maintaining a low computational load. The equivalent circuit includes:

- A state-of-charge (SOC) controlled voltage source, whose value ranges from a minimum discharge voltage (V_d) to a maximum charge voltage (V_f).

- A series resistor (R_s) that models the internal resistive losses.

- A parallel network consisting of a resistor (R_p) and a capacitor (C), which captures the polarisation effects and dynamic response of the cell.

- An internal calculation of the SOC from the net current supplied or absorbed, considering a total capacity (Cap) and a configurable initial state ($InitSOC$).

The model is implemented in Modelica (Figures 4.6, 4.7) from basic components and algorithmic equations for the SOC. Additionally, current limitation has been implemented to constrain the maximum current that the actual physical device would be capable of providing.

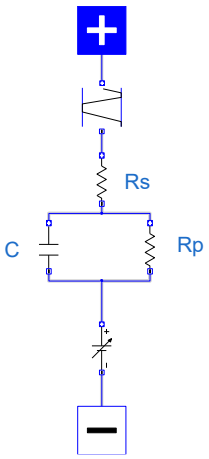

Figure 4.6: Battery Model Component Diagram.

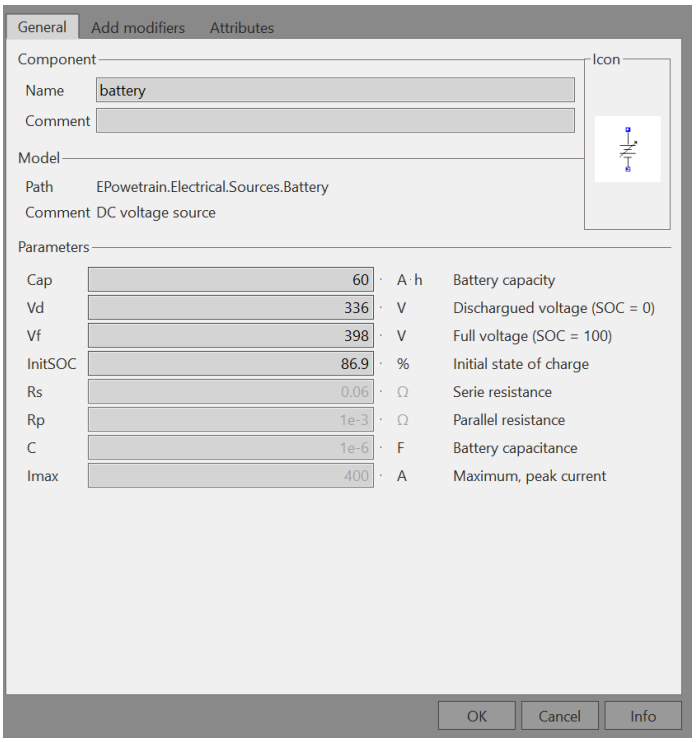

Figure 4.7: Battery Model Parametrization Interface.

4.6.2 ElectricConverter

The *ElectricConverter* model implements an idealised representation of a power converter (such as an inverter or a DC-DC converter) by means of a simple linear relationship controlled by a modulating signal as shown in chapter 3.3.

$$V_{out} = DutyCycle \cdot V_{in} \tag{4.1}$$

This approach avoids Modeling the individual PWM switching cycles, which reduces the computational burden and avoids introducing discontinuities that can degrade the numerical performance of the simulator. This type of model is especially useful for long duration driving simulations such as UDDS. To ensure the ideal behaviour of the converter, the following restriction has been introduced in the power calculation. Forcing the power balance to be fulfilled.

$$Pw_{In} + Pw_{Out} = V_{In} \cdot I_{In} + V_{Out} \cdot I_{Out} = 0 \tag{4.2}$$

Where the current flow is reversed depending on the mode of operation:

$$I_{in} = \begin{cases} -I_{inputPin} & if \quad DutyCycle \geq 0 \quad Discharging\ mode \\ I_{inputPin} & if \quad DutyCycle < 0 \quad Charging\ mode \end{cases} \tag{4.3}$$

This inversion of current flows has been introduced to control the charging mode (regenerative braking) where a negative *DutyCycle* implies a power flow to the battery (charging) and a positive *DutyCycle* symbolises a power flow from the battery to the engine (discharging). Figure 4.8 below shows the source code of the modelica implementation of this submodule.

```
model ElectricConverter
  "DC/DC ideal power converter"
  Interfaces.PosPin P_In ;
  Interfaces.NegPin N_In ;
  Interfaces.PosPin P_Out ;
  Interfaces.NegPin N_Out ;
  Interfaces.InPort DutyCycle ;
  input SI.Voltage V_In=P_In.v - N_In.v;
  output SI.Voltage V_Out=P_Out.v - N_Out.v;
  input SI.Current I_In;
  // Input port current (Source side)
  output SI.Current I_Out=P_Out.i;
  // Output port current (Load side)

  SI.Power PwIn=V_In*I_In;
  SI.Power PwOut=V_Out*I_Out;

  SI.Energy EBalance(start=0);
equation

  der(EBalance) = PwIn + PwOut;

  I_In = if DutyCycle >= 0 then P_In.i else -P_In.i;

  // Set V_Out
  V_Out = DutyCycle*V_In;

  PwIn + PwOut = 0;

  //Currents balance
  P_In.i + N_In.i = 0;
  P_Out.i + N_Out.i = 0;

end ElectricConverter;
```

Figure 4.8: ElectricConverter Implementation in Modelica.

4.6.3 DCMotor

The *DCMotor* model (Figures 4.9, 4.10) is the Modelica implementation of a permanent magnet direct current (PMDC) motor whose mathematical model was discussed in chapter 3.2. For the implementation, the motor is composed of 3 elements: Resistance R_1 and inductance L_1 represent the input impedance to the motor while the *BackEMF* submodel integrates the electromechanical phenomena that allow the transformation from the electrical to the mechanical domain.

$$V_p - V_n = R_1 \cdot I + L_1 \cdot \frac{dI}{dt} + V_{emf} = R_1 \cdot I + L_1 \cdot \frac{dI}{dt} + K_e \cdot \omega \tag{4.4}$$

$$T_m = K_t \cdot I = T_{load} + B \cdot w + J \cdot \frac{d\omega}{dt} \tag{4.5}$$

The *BackEMF* submodule deals with the conversion of electric current to torque and vice versa. For this purpose, it integrates the counter-electromotive force equations discussed in chapter 3.2.

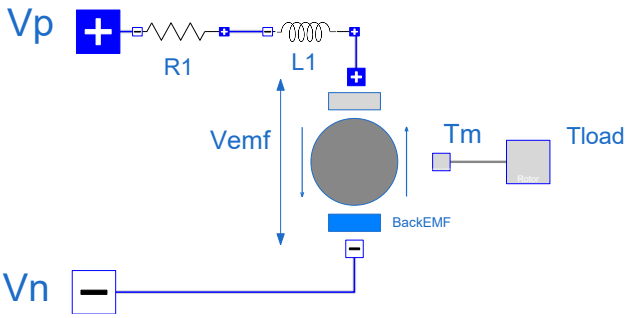

Figure 4.9: DC Motor Modelica Implementation.

```
model BackEMF
  "Counter-electromotive force"
  Interfaces.PosPin Pp ;
  Interfaces.NegPin Np ;
  Interfaces.MechanicalAxis mechanicalAxis
    ;
  parameter Real Ke(unit="V.s/rad") = 0.0064
    "Back emf constant";
  parameter Real Kt(unit="N.m/A") = 0.0065
    "Torque constant";
  parameter SI.RotationalDampingConstant bm=4.121*1e-6
    "Friction constant";
  parameter Modelica.Units.SI.Inertia J=3.87*1e-7;
  Modelica.Units.SI.Voltage Vemf;
  Modelica.Units.SI.Current i=Pp.i;
  Modelica.Units.SI.Torque Te "Electrical torque";
  Modelica.Units.SI.Torque Tb "Friction torque";
  Modelica.Units.SI.Torque Tload=mechanicalAxis.T
    "Axis torque";
  Modelica.Units.SI.Angle Phi=mechanicalAxis.Phi;
  Modelica.Units.SI.AngularVelocity w=der(Phi);
  constant Real pi=Modelica.Constants.pi;

equation
  Pp.i + Np.i = 0 "Currents balance";
  Vemf = Ke*w;
  Vemf = Pp.v - Np.v;
  Te = i*Kt;
  Tb = bm*w;
  Te - Tb - Tload = J*der(w);


end BackEMF;
```

Figure 4.10: BackEMF Submodel Source Code.

4.7 Mechanical

The Mechanical package (Figure 4.11) provides essential models to represent both rotational dynamics and longitudinal displacement of the electric vehicle. The models were designed to maintain physical compatibility with the rest of the library architecture, coherently integrating the mechanical domain with the electrical and control components. This section extends the previously introduced rotational type models and incorporates the key models used to represent the linear motion of the vehicle, in particular the chassis (*BodyFrame1DOF*), the wheel (*Wheel*) and the effect of the terrain slope (*Slope*).

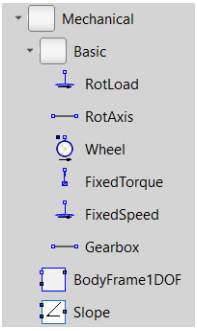

Figure 4.11: Mechanical Subpackage Modules.

4.7.1 Wheel

The *Wheel* model acts as an ideal transformer that converts the mechanical torque T applied to the axle of a wheel into a linear force F, transmitted to the vehicle body through a translational interface.

The relationships used are:

- Vehicle linear velocity:

$$v = 2 \cdot \pi \cdot R \cdot \omega \tag{4.6}$$

Where R and ω represents the wheel radius and angular speed respectively.

- Transmitted traction force balance:

$$T_{axis} + f_s \cdot \omega + (J + M \cdot R^2) \cdot \frac{d\omega}{dt} = 0 \tag{4.7}$$

With T_{axis} is the shaft provided torque, f_s the dynamic viscosity friction coefficient. The term:

$$(J + M \cdot R^2) \cdot \frac{d\omega}{dt} \tag{4.8}$$

Models the inertia forces resulting from both linear and angular acceleration of the wheel. The rotational component corresponds to a disc inertia:

$$F_{rot} = J \cdot \frac{d\omega}{dt} \tag{4.9}$$

With J as the inertial coefficient of the disc(wheel). The linear acceleration inertia can be transformed in an equivalent applied torque on the wheel axis (Figure 4.12) if we consider that the mass works in opposition to the mass acceleration.

$$F_I = M \cdot \frac{dv}{dt} = M \cdot R \cdot \frac{d\omega}{dt} \tag{4.10}$$

$$T_{F_I} = F_I \cdot R = M \cdot R^2 \cdot \frac{d\omega}{dt} \tag{4.11}$$

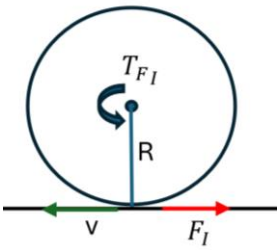

Figure 4.12: Wheel Model Lineal Inertia Force.

The total torque resulting from the inertial forces then results:

$$T_I = T_{F_I} + T_{rot} = J \cdot \frac{d\omega}{dt} + M \cdot R^2 \cdot \frac{d\omega}{dt} = (J + M \cdot R^2) \cdot \frac{d\omega}{dt} \tag{4.12}$$

The model also integrates a steady state mode, where the steady state friction force is modelled when the velocity and accelerations are close to zero. Instead of assigning an equation on ω directly, we work on its derivative so that the continuity of the variable in the changing modes of operation can be respected. Avoiding chattering problems.

$$\frac{d\omega}{dt} = \begin{cases} -10^3 \cdot \omega & |\omega| < eps \\ \dfrac{T_{axis} + f_s \cdot \omega}{(J + M \cdot R^2)} & |\omega| \geq eps \end{cases} \tag{4.13}$$

Figure 4.13 illustrates the Wheel model implementation below.

```
model Wheel "Wheel load model"
  constant Real pi=Constant.pi;
  Interfaces.MechanicalAxis Axis ä ;
  parameter SI.Inertia J=10 "Inertial load";
  parameter SI.Length R=0.3 "Radius";
  parameter Real fs(
    unit="N.m.s/rad",
    min=0) = 8.5e-6 " Dynamic viscosity";
  parameter SI.Mass M=1200 "Vehicle mass";
  constant Real eps=1e-3;
  SI.AngularVelocity w;
  SI.Velocity V(start=0);
  Interfaces.OutPort Vwheel ä ;
equation
  der(Axis.Phi) = w;

  V = 2*pi*R*w;

  if abs(w) < eps then
    //Stacionary state
    der(w) = -1e3*w;
  else
    Axis.T + w*fs + (J + M*R^2)*der(w) = 0;

  end if;
  Vwheel = V;
  ä

end Wheel;
```

Figure 4.13: Wheel Model Source Code.

4.7.2 BodyFrame1DOF

BodyFrame1DOF represents the vehicle body as a point mass with only one degree of freedom (longitudinal displacement). It is the model that simulates the motion of the vehicle under the action of the driving and resisting forces (Figure 4.14).

The model is implemented as a body of mass m subjected to a net force according to the equation:

$$F_m = F_I + F_d + F_r + F_g \tag{4.14}$$

The calculated forces correspond to those given in section 3.5:

$$F_m = \frac{T_{In}}{R} \tag{4.15}$$

Where T_{in} is the torque provided by the motor and R *the wheels radius*.

$$F_I = m \cdot \frac{dv}{dt} \tag{4.16}$$

With v as the vehicle's linear velocity.

$$F_d = \frac{1}{2} \cdot \frac{C_d \cdot \rho \cdot v^2}{A_f} \tag{4.17}$$

In which C_d, A_f represents the vehicle's drag coefficient and front area respectively and ρ the media fluid (air) density.

$$F_r = C_r \cdot m \cdot g \cdot cos(\alpha) \tag{4.18}$$

Where C_r is the tyres' rolling friction coefficient and g the gravity constant.

$$F_g = m \cdot g \cdot sen(\alpha) \tag{4.19}$$

This model allows to simulate the basic physics of the vehicle as well as to take into consideration the effects of the terrain in terms of slope changes on the stresses required to the vehicle's powertrain.

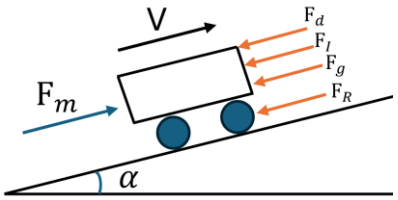

Figure 4.14: BodyFrame1DOF Forces Distribution.

```
model BodyFrame1DOF
  "1 degree of freedom body frame"

  Interfaces.MechanicalAxis TorqueIN ;
  output Interfaces.OutPort V "Vehicle speed"
    ;
  input Interfaces.InPort Alpha "Terrain slope"
    ;
  constant SI.Acceleration g=Modelica.Constants.g_n;
  constant Real pi=Modelica.Constants.pi;
  parameter SI.Length R(min=0.01) = 0.25 "Wheel radius";
  parameter SI.Mass M=1500 "Vehicle mass";
  parameter SI.Area Af=2 "Vehicle front area";
  parameter Real Cd(min=0) = 0.248 "Vehicle drag coef.";
  parameter Real Cr(min=0) = 0.01
    "Tyres rolling resistance coef.";
  parameter SI.Density rho(displayUnit="kg/m3") = 1.2
    "Air density";

  SI.Force F "Powertrain provided force";
  SI.Force Fd "Drag force";
  SI.Force Fr "Rolling resistance";
  SI.Force Fg "Weight poryected force";
  SI.Force Fi "Inertial force";

equation
  Fd = 0.5*Cd*Af*rho*(V^2)*sign(V);
  Fr = M*g*cos(Alpha*pi/180)*Cr;
  Fg = M*g*sin(Alpha*pi/180);
  Fi = M*der(V);

  F - Fd - Fr - Fg - Fi = 0;
  F = -TorqueIN.T/R;
  der(TorqueIN.Phi) = V/(2*pi*R);

  

end BodyFrame1DOF;
```

Figure 4.15: BodyFrame1DOF Source Code.

4.7.3 Slope

The *Slope* model, defined in the *Mechanical* subpackage, allows the instantaneous terrain slope to be calculated from a height profile and the vehicle position. It is an auxiliary component that converts two input signals, height and displacement into an output signal representing the slope angle of the terrain.

$$\alpha = \arcsin\left(\frac{\frac{dH}{dt}}{\frac{ds}{dt}}\right) \tag{4.20}$$

This model is especially useful in realistic simulations with routes extracted from topographic profiles, as it allows the effect of variable slopes to be dynamically incorporated without the need for manual coding. As show in Figure 4.16, this calculation is performed only after instant $t > 0$, to avoid division by zero in the initialisation.

```
model Slope "Terrain Slope"

  Interfaces.InPort H "Current Heigth" ∃ ;
  Interfaces.OutPort Alpha "Slope" ∃ ;

  Interfaces.InPort s "Current displacement"
    ∃ ;


equation
  when (time > 0) then
    Alpha = asin(der(H)/der(s));
  end when ∃ ;
  ∃
end Slope;
```

Figure 4.16: Slope Source Code.

This model does not represent a physical force directly but generates a slope angle that can be used by other blocks (Figure 4.17), such as the BodyFrame1DOF model, to apply the corresponding gravitational and tyre rolling forces.

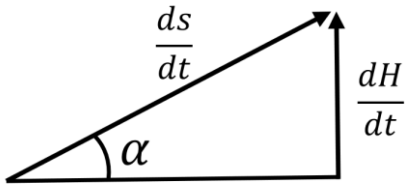

Figure 4.17: Slope Angle Representation.

4.8 Control

The *Control* package (Figure 4.18) contains the necessary blocks to implement control strategies within the electric drive system. Its purpose is to dynamically regulate the behaviour of actuators (such as electric motors or inverters) based on references and feedback signals. Although a simple structure has been kept, the models included are flexible enough to address common tasks in energy management and speed control.

Modelica's modular and acausal architecture allows these controllers to be easily coupled to different physical components of the system, with generic connectors carrying continuous variables.

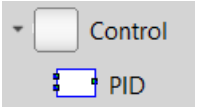

Figure 4.18: Control Package Module.

PID

This block implements a continuous PID controller with extended functionalities such as dead zone, output saturation limits and anti-windup. It allows to adjust the three classical PID control terms, proportional K, Integral I and derivative D. The model is implemented in differential form, with explicit equations for the calculation of the filtered output (Out) and the integration of the accumulated error ($IntEr$). The gross output (Aux_{Out}) of the controller is calculated as a classical PID:

$$Aux_{Out} = K \cdot e + D \cdot \frac{de}{dt} + I \cdot IntEr \tag{4.21}$$

Where K, D and I are the proportional, derivative and integral constants of the controller respectively. The input error is calculated form the difference between the reference value and the actual feedback signal input:

$$e = Ref - In \tag{4.22}$$

The term $IntEr$ corresponds to the error integration. However, in Modelica it is more convenient to define the error as the derivative of its integral.

$$IntEr = \int_0^t e \cdot d\tau \rightarrow \frac{dIntEr}{dt} = e \tag{4.23}$$

The controller includes two advanced mechanisms to adapt the response of the controller. The first of these is the implementation of a dead zone. When this option is activated, the controller suppresses the output action and prevents the accumulation of the integral error in a near-zero environment. This approach is useful to avoid unnecessary oscillations or reactions to small disturbances. If the enable condition is met:

$$DeadZone = True\ and\ |Aux_{Out}| < \epsilon \tag{4.24}$$

Then the output and error integration are suppressed.

$$\frac{dOut}{dt} = -Smt \cdot Out \tag{4.25}$$

$$\frac{dIntEr}{dt} = 0 \tag{4.26}$$

Where ϵ is the width of the dead zone while Smt is a smoothing factor to decay the output to zero without introducing discontinuities. The further implemented functionality is the limitation of the controller output. In addition, an anti-windup mechanism has been considered to avoid integral error accumulation when the controller output is saturated. These consider three working modes:

- If the auxiliary output exceeds the upper limit:

$$Aux_{Out} \geq Max \rightarrow \begin{cases} \frac{dOut}{dt} & = Smt \cdot (Max - Out) \\ \frac{dIntEr}{dt} & = 0 \end{cases} \tag{4.27}$$

- When the auxiliary output falls below the lower limit:

$$Aux_{Out} \leq Min \rightarrow \begin{cases} \frac{dOut}{dt} & = Smt \cdot (Min - Out) \\ \frac{dIntEr}{dt} & = 0 \end{cases} \tag{4.28}$$

- In any other case the output is adjusted to the previously calculated raw performance.

$$Min < Aux_{Out} < Max \rightarrow \begin{cases} \frac{dOut}{dt} & = Smt \cdot (Aux_{Out} - Out) \\ \frac{dIntEr}{dt} & = e \end{cases} \tag{4.29}$$

This approach avoids undesired behaviour when the actuator cannot follow the setpoint signal, retaining a more realistic and stable response in the presence of saturations in addition to ensuring the continuity of the output signal. In Figures 4.19 , 4.20 the internal source code and parametrization model interface are displayed.

```
model PID "Proportional-Integral-Derivative controller"

  Interfaces.OutPort Out ;
  Interfaces.InPort In ;
  Interfaces.InPort Ref ;

  parameter Boolean LimitOut=false
    ;
  parameter Real Max=0;
  parameter Real Min=0;
  parameter Boolean DeadZone=false
    ;
  parameter Real eps(min=0) = 1e-9 "Dead zone range";
  parameter Real K=1;
  parameter Real I=0;
  parameter Real D=0;
  parameter Real Smt=1e6 "Smooth factor";

  Real AuxOut(start=0);
  Real Error(start=0);
  Real IntEr(start=0);

equation

  AuxOut = K*Error + D*der(Error) + I*IntEr;
  Error = Ref - In;

  // DEADZONE
  if DeadZone == true and noEvent(abs(AuxOut) < eps)
       then
    der(Out) = -Smt*Out;
    der(IntEr) = 0;
  elseif LimitOut == true then
    //ANTIWINDUP
    if noEvent(AuxOut >= Max) then
      der(Out) = Smt*(Max - Out);
      der(IntEr) = 0;
    elseif noEvent(AuxOut <= Min) then
      der(Out) = Smt*(Min - Out);
      der(IntEr) = 0;
    else
      der(Out) = Smt*(AuxOut - Out);
      der(IntEr) = Error;
    end if;
  else
    der(Out) = Smt*(AuxOut - Out);
    der(IntEr) = Error;
  end if;

end PID;
```

Figure 4.19: PID Controller Modelica Implementation.

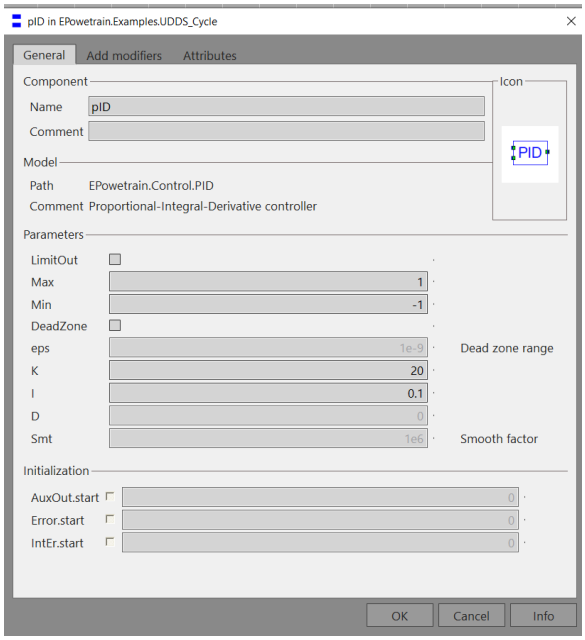

Figure 4.20: PID Block Interface.

4.9 Sensors

The *Sensors* package (Figure 4.21) contains blocks ideal for measuring key physical variables in an electrical powertrain, such as voltage, current, angular velocity and position. These sensors are used for two purposes: controller feedback (such as the PID block) and recording simulation results.

All models are implemented as ideal sensors, without delay, offset or noise, which simplifies their integration into functional experiments. However, their modular structure allows them to be easily extended with dynamics or errors if more realistic simulations are desired.

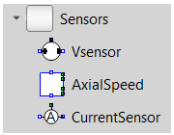

Figure 4.21: Sensors Package Components.

4.9.1 Vsensor – Electrical Voltage Sensor

The *Vsensor* model (Figure 4.22) measures the electrical potential difference between the two pins (p,n) of the system and delivers the result through a continuous output connector (outPort), which can be connected to control or display blocks.

$$v = v_p - v_n \tag{4.30}$$

$$out_{Port} = v \tag{4.31}$$

This sensor is non-intrusive: it imposes zero current on the terminals, which makes it suitable for idealised measurements.

$$I_p = I_n = 0 \tag{4.32}$$

```
model Vsensor "Voltage measurement sensor"

  EPowetrain.Interfaces.PosPin p ;
  EPowetrain.Interfaces.PosPin n ;

  Modelica.Units.SI.Voltage v;
  Interfaces.OutPort outPort ;
equation
  p.i = 0;
  n.i = 0;
  v = p.v - n.v;
  outPort = v;

end Vsensor;
```

Figure 4.22: Vsensor Source Code.

4.9.2 CurrentSensor – Electric Current Sensor

This block measures the current flowing between two electrical nodes. The connection is made via a positive pin (P) and a negative pin (N). The sensor imposes equal potential between the two terminals, so it does not introduce voltage drops.

$$v_p = v_n \tag{4.33}$$

$$I_p + I_n = 0 \tag{4.34}$$

The unit of measurement I_{meas} , is the current passing through and out of the component, i.e. the negative terminal.

$$I_n = I_{meas} \tag{4.35}$$

This ideal sensor assumes perfect coupling and no losses. Its output can be used to assess energy consumption, estimate battery state of charge or feed power control algorithms. Figure 4.23 illustrates the Modelica's implementation of this module.

```
model CurrentSensor "Current measurement sensor"

  Interfaces.PosPin P ;
  Interfaces.NegPin N ;
  Interfaces.OutPort Imeas ;


equation
  P.v = N.v;
  N.i = Imeas;
  P.i + N.i = 0;


end CurrentSensor;
```

Figure 4.23: CurrentSensor Source Code.

4.9.3 AxialSpeed – Angular Position and Velocity Sensor

The *AxialSpeed* model measures the position θ_m and angular velocity ω_m of the mechanical axis of a machine and also calculates the electrical equivalent values (θ_e, ω_e) considering the number of pole pairs N. It is connected between two rotational axes ($Axis_{In}$ and $Axis_{Out}$), without introducing friction or inertia.

$$\theta_e = N \cdot \theta_m \tag{4.36}$$

$$\omega_m = \frac{d\theta_m}{dt} \tag{4.37}$$

$$\omega_e = \frac{d\theta_e}{dt} \tag{4.38}$$

In addition, the model includes an electric cycle reset:

$$if\ \theta_e \geq 2\pi \rightarrow reinit(\theta_e, 0) \tag{4.39}$$

This condition prevents indefinite growth of the electrical angular position, facilitating its use in cyclic logic as a switching or synchronisation control (Figure 4.24).

```
model AxialSpeed
  "Angular position and speed measurement sensor"

  Interfaces.MechanicalAxis Axis_In ;
  Interfaces.MechanicalAxis Axis_Out ;
  Interfaces.OutPort Wm ;
  Interfaces.OutPort Th_m ;
  Interfaces.OutPort We ;
  Interfaces.OutPort Th_e ;
  parameter Integer N(min=1) "Pole pairs";
  constant Real pi=Constant.pi;

equation

  Th_e = N*Axis_In.Phi;
  Th_m = Axis_In.Phi;

  when (Th_e >= 2*pi) then
    reinit(Th_e, 0);
  end when;

  Wm = der(Th_e);
  We = der(Th_m);

  connect(Axis_In, Axis_Out) ;


end AxialSpeed;
```

Figure 4.24: AxialSpeed Source Code.

4.10 Interfaces and Interoperability

Interoperability between components within the EPowertrain library is achieved through a coherent set of interfaces, developed under the acausal Modeling paradigm of the Modelica language. These interfaces ensure physical compatibility and simplify the integration of new elements, even when they belong to different physical domains.
The library defines a set of standard connectors, including:

- **Electrical interfaces**: Based on PosPin and NegPin, these connectors define voltage and current as ‘stress’ and ‘flow’ variables, respectively. They are used consistently on all electrical components to ensure proper signal and power transmission.

- **Mechanical interfaces**: The MechanicalAxis connector enables the transmission of torque and angular velocity between elements such as motors, shafts and loads.

- **Control interfaces**: Control signals are handled by connectors such as IO_Port, InPort and OutPort, which carry scalar or logical variables (Boolean, Real). This facilitates the integration of control strategies without creating rigid couplings between components.

Although the library has been designed primarily for internal use, the definition of interfaces is generic and compatible with other Modelica libraries. This allows external components to be integrated into simulations with minimal adaptations, provided that compatible connector types are used.

4.11 Conclusions

This chapter has presented the structured implementation of the EPowertrain library, developed in the Modelica environment for the simulation of electric power trains. The internal architecture of the library, organized in functional subpackages grouping electrical, mechanical, control, sensor and interface components, has been detailed. This modular structure facilitates the reuse, extension and understanding of the models by other users and developers.

Each subcomponent has been designed following acausal modeling and object-oriented programming principles, taking advantage of the capabilities of the Modelica language to represent physical interactions through declarative equations. This has made it possible to define configurable components that can be connected to each other in a physically consistent way, keeping compatibility between interfaces and preserving the conservation of quantities such as energy, current or mechanical stress.

At the same time, we have sought to parameterize as much as possible the calibration of the components in order to facilitate reuse and integration in more complex systems.

5 EPowertrain Library Validation

5.1 Introduction

Once the individual models and the general architecture of the EPowertrain library have been developed, it is necessary to verify that their simulated behaviour adequately matches the expected physical behaviour. This chapter presents the validation process carried out on the different components and configurations of the library, in order to evaluate their accuracy, physical consistency and applicability in real electric vehicle simulation contexts. First, the individual validation of the following key components is described:

- DC motor
- Battery
- DC-DC Electric Converter

Following this, the validation of the complete system is introduced It was first tested in a synthetic test under the UDDS driving [1] cycle followed by the use of real driving data of a real vehicle, the BMW i3 [2], to compare real versus simulated results.

The main goal of this chapter is to demonstrate that the developed library allows to reproduce, with fidelity and computational efficiency, the energy behaviour of an electric power train under realistic dynamic conditions, as well as being a valid tool for consumption studies, control strategies and architecture comparison.

5.2 Validation of Individual Components

Before proceeding to the validation of complete systems, individual validation of the main components developed in the EPowertrain library was carried out. This process was essential to ensure that each model presented a behaviour consistent with its theoretical description and met the established functional requirements.

5.2.1 DC Motor

The dynamic response of the model was checked using parameter identified based on Moments and Pasek techniques, following the procedures described in [36] .The relationship between current, torque and angular velocity under voltage step excitations was evaluated.

To validate both methods, the authors performed dynamic tests by applying a voltage step excitation to an independently excited DC motor. A voltage step varying from 60 V to 248 V was applied to the motor armature, and the current and angular velocity responses were recorded. Test parameters can be checked in Table 5.1.

Table 5.1: Comparison of DC Motor Parameters Identified by Pasek's Method and Moments Method.

Parameter	**Symbol**	**Pasek's**	**Moments**	**Unit**
Armature resistance	R_a	30.9	30.9	Ω
Armature inductance	L_a	0.438	0.803	H
Torque and back-EMF constant	K	1.323	1.323	$N \cdot m/A$
Rotor inertia	J	0.0036	0.0031	$kg \cdot m^2$
Viscous friction coefficient	f	0.0005	0.0005	$N \cdot m \cdot s/rad$
Static torque T_{st}	Tst	0.128	0.128	$N \cdot m$

During these tests, the following experimental values were observed:

- Armature current went from 0.113 A in the initial state to 0.167 A in the final state after excitation.
- The angular velocity of the rotor increased from 400 rpm to 1745 rpm.
- The maximum instantaneous current was recorded at approximately 0.026 seconds after the start of the step. With a peak of approximately 4.5 A in the case of Pasek's estimation and 4 A in the moments' estimation.

In the case of our simulation (Figure 5.1), we obtain a steady-state speed of approximately 1772 rpm for both methods, as well as peak currents of 4.5 and 3.985 A using the Pasek and moments models respectively. These results allow us to conclude that the model developed is capable of accurately reproducing the dynamic behaviour of the motor under controlled excitation conditions, thus validating its use in the context of energy analysis of electric kinematic chains.

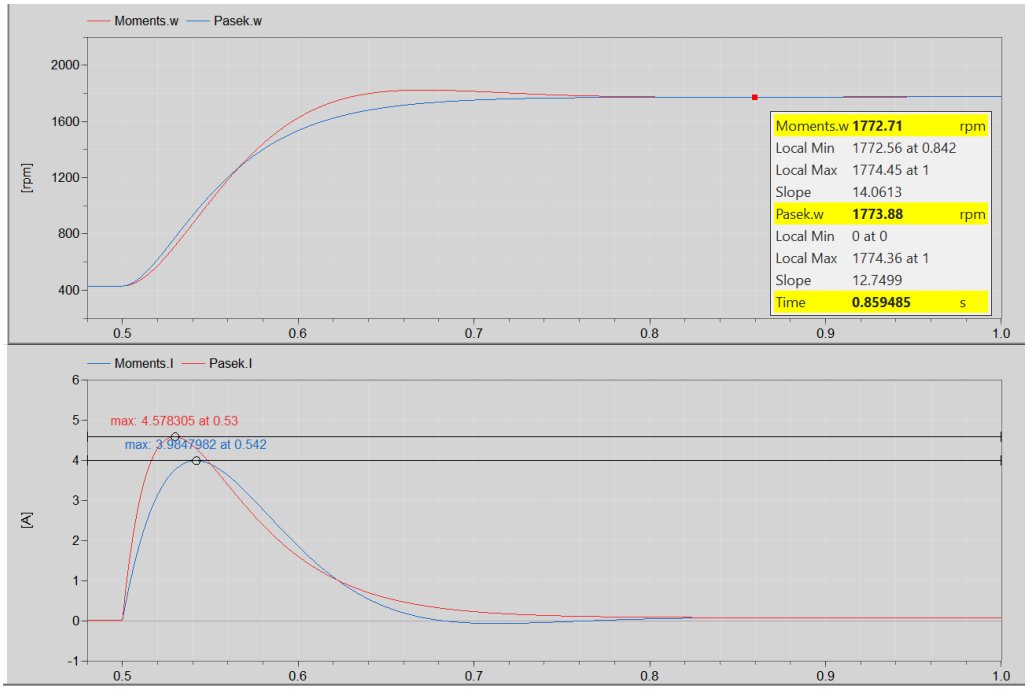

Figure 5.1: Modelica Results of Pasek and Moments Estimated Model Simulations.

5.2.2 Battery

Before proceeding to the validation of the equivalent battery model proposed in this library, it is necessary to have experimental data to compare the response of the model with the real behaviour of the physical system. For this purpose, an urban driving project with a BMW i3 dataset will be used [2].

Within the set of available trips, the file named TripB14 has been specifically chosen, since it presents one of the largest state-of-charge (SOC) variations between start and end of the trip (Figure 5.2). This feature is particularly relevant for validation, as it allows to evaluate the ability of the model to reproduce not only the instantaneous voltage variations, but also the cumulative evolution of the SOC over time under dynamic conditions.

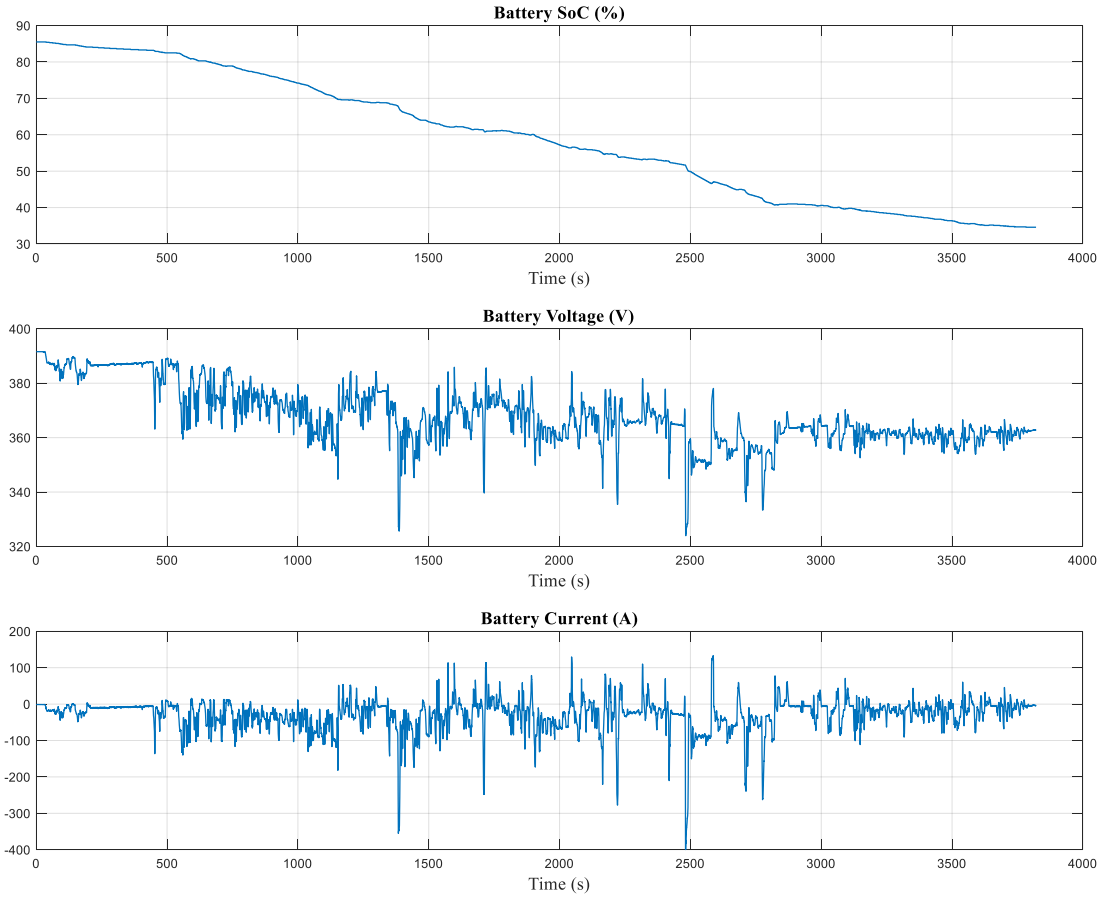

Figure 5.2: TripB14 Battery Dataset. From Top to Bottom: Soc, Voltage, Current Consumption.

The battery consists of a RC1 model whose characteristic equations have already been specified in chapter 3.4.

$$SOC(t) = SOC_{init} - \frac{1}{Q}\int_0^t I(\tau)\cdot d\tau \tag{5.1}$$

$$V_{oc}(SOC) = V_d + (V_f - V_d) \cdot SOC \tag{5.2}$$

$$V_{Batt} = V_{OC} - I \cdot R_s - V_{RC} \tag{5.3}$$

Where V_{RC} is the voltage drop in the parallel RC network:

$$C \cdot \frac{dV_{RC}}{dt} = \frac{V_{OC} - V_{RC}}{R_p} - I \tag{5.4}$$

This model has been previously described by other authors as a system in state space [37].

$$\frac{dV_{RC}}{dt} = \frac{1}{R_p \cdot C} \cdot V_{RC} + \frac{1}{C} \cdot I \tag{5.5}$$

$$V_{Batt} = V_{RC} + R_S \cdot I + V_{oc} \tag{5.6}$$

There is not available data for the estimation of open circuit voltage; however, it is possible to estimate it by taking points where the current is as constant and reduced as possible to minimize the layer and resistive effects of the battery. For this purpose, we have chosen initial and final points of the driving cycle where the vehicle is at rest.

$$V_{Batt} \approx V_{oc} \tag{5.7}$$

Under these assumptions, we can make a linear estimation of the open circuit voltage of the battery as a function of its state of charge such that:

$$\begin{bmatrix} V_{Batt_0} \\ V_{Batt_f} \end{bmatrix} = \begin{bmatrix} 1 - SOC_0 & SOC \\ 1 - SOC_f & SOC \end{bmatrix} \begin{bmatrix} V_d \\ V_f \end{bmatrix} \tag{5.8}$$

$$\begin{bmatrix} 391.6 \\ 362.8 \end{bmatrix} = \begin{bmatrix} 0.145 & 0.855 \\ 0.654 & 0.346 \end{bmatrix} \begin{bmatrix} V_d \\ V_f \end{bmatrix} \tag{5.9}$$

This gives us an estimate for the full and depleted battery tension of:

$$V_d \approx 343.22 \text{ V} \tag{5.10}$$

$$V_f \approx 399.80 \text{ V} \tag{5.11}$$

To estimate the parameters Rs, Rp and C it is possible to take advantage of the dataset's sample space to perform an estimation using the System Identification MATLAB toolbox. Once we have an estimate of V_{oc} , we can perform several estimates for different SoC.

Table 5.2: Battery Parameter Estimation.

SoC	R_s	R_p	C
10 %	0.1159 Ω	$1.337 \cdot 10^{-4}$ Ω	$1.81 \cdot 10^{5}$ F
25 %	0.0896 Ω	$6.8102 \cdot 10^{-4}$ Ω	$1.2497 \cdot 10^{4}$ F
50 %	0.1051 Ω	$1.7834 \cdot 10^{-4}$ Ω	$9.5329 \cdot 10^{3}$ F
75 %	0.1093 Ω	$6.9485 \cdot 10^{-4}$ Ω	$2.008 \cdot 10^{4}$ F
100 %	0.1183 Ω	$2.2333 \cdot 10^{-4}$ Ω	$1.4817 \cdot 10^{5}$ F

It is worth noting the variability of the parameter estimates (Figure 5.3) depending on whether the state of charge is close to extreme values of unloaded or full or whether it is in more average values. This may be due either to the characteristics of the battery itself or to the fact that calculations have been made for SoC values outside the sample space (it must be pointed out that in the dataset the SoC of the battery is between 85.5% and 34.6%).

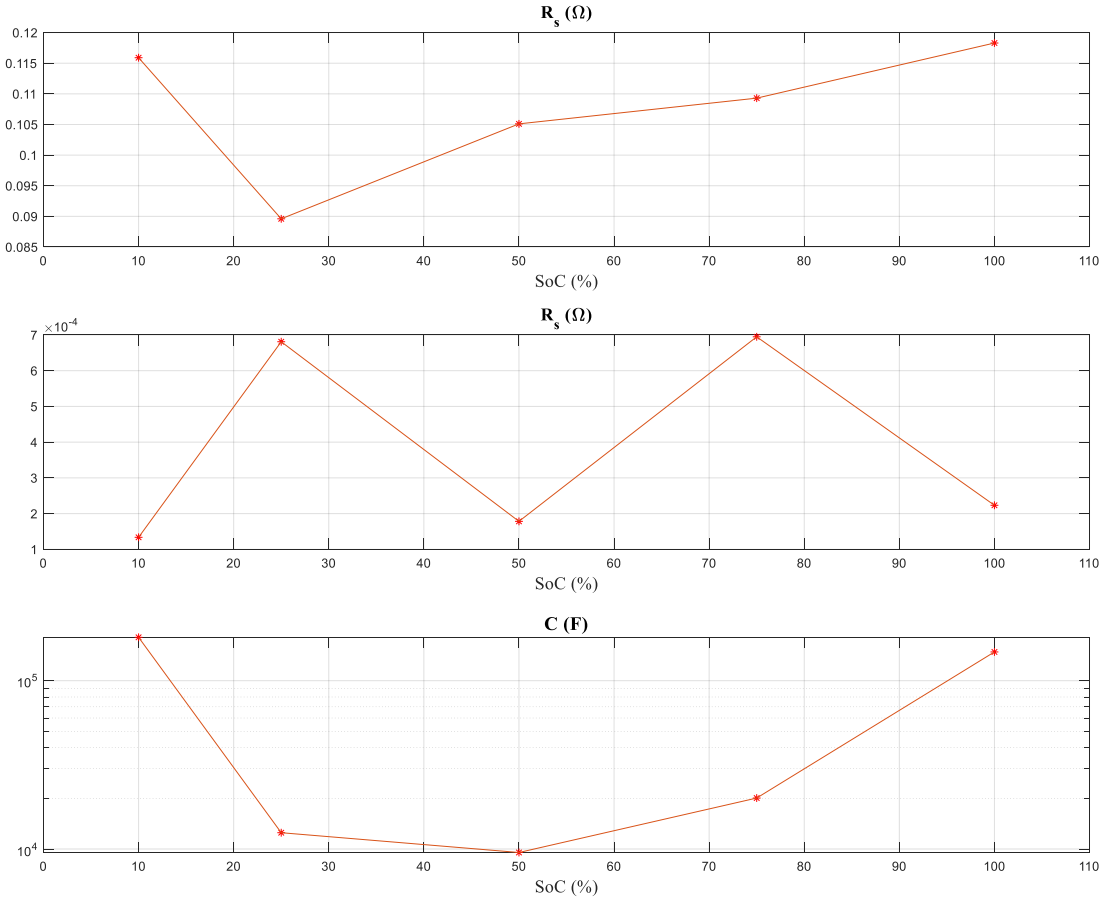

Figure 5.3: Battery Parameter Estimation. From Top to Bottom: Series Resistance, Parallel Resistance, Parallel Capacitance.

For this reason, it might be preferable to be conservative and choose the estimated parameters for a 50% state of charge in a first approximation. Once a parameterization that characterizes the battery has been estimated, it is necessary to check that the battery behaviour matches reality. For this purpose, as will be explained later in chapter 5.4, the TripA01 [2] driving cycle has been used as validation data for the complete system and comparing the simulated model response against real terminal voltage and state-of-charge data. As shown in Figure 5.4, the model reproduces with high fidelity the SOC

evolution along the path, with a practically negligible cumulative error. This confirms that the integration of the net current is consistent and that the model correctly represents the load balance under dynamic conditions.

As for the terminal voltage, the model adequately captures the general trend and rapid variations due to current transients, although it shows some point deviations in the steeper areas. These differences are attributable, in part, to the simplicity of the first-order RC model employed, which does not account for effects such as hysteresis, thermal dependence of the parameters or nonlinearity of the open-circuit curve (OCV).

Therefore, the proposed model is suitable for applications focused on energy consumption analysis, SOC estimation or evaluation of control strategies in early design stages.

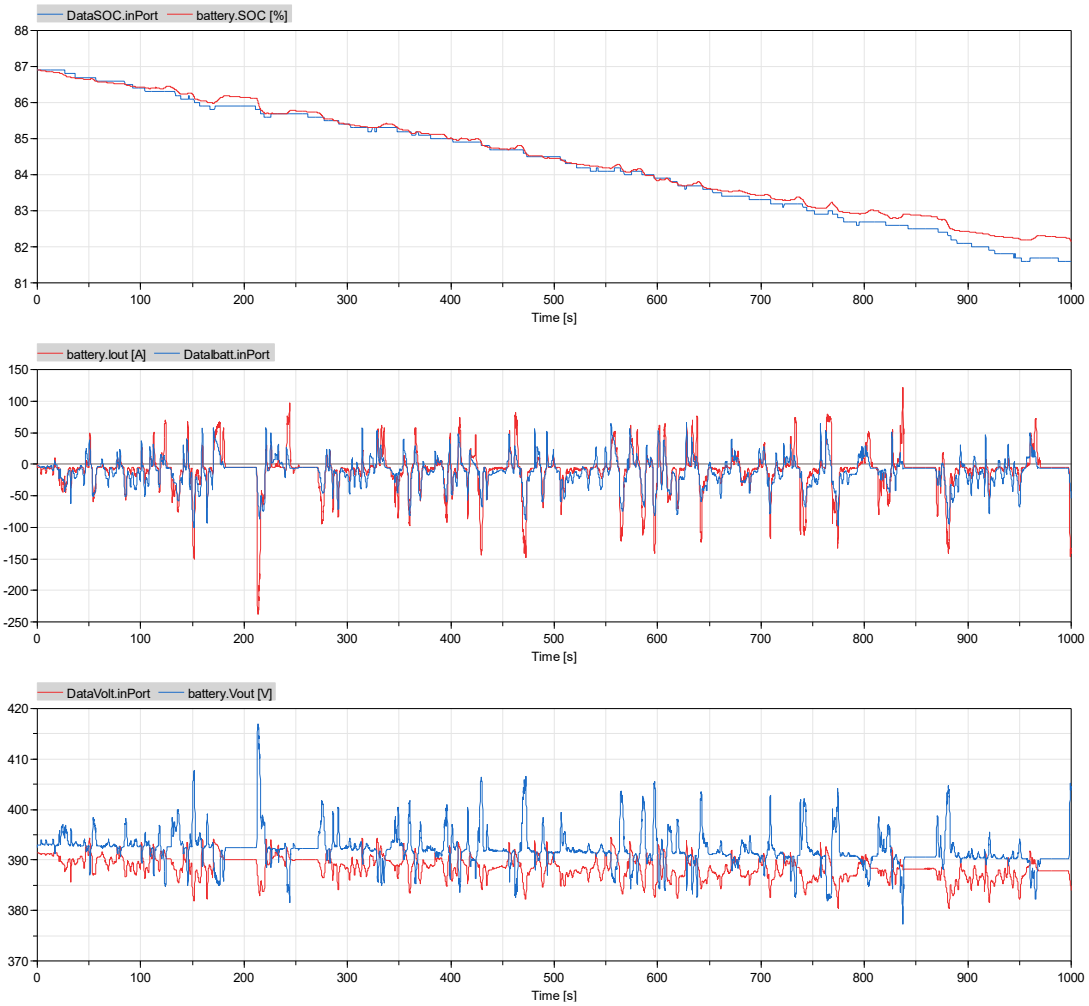

Figure 5.4: Battery Data and Simulation Comparison. From Top to Bottom: SoC, Current Consumption, Battery Voltage.

Table 5.3: Battery Simulation Parameters.

Parameter	Symbol	Value
Series resistance	R_s	$0.1051\ \Omega$
Parallel resistance	R_p	$1.7834\ \mathrm{e}-4\ \Omega$
Capacitor capacitance	C	$9.5329 \cdot 10^{-3}\ F$
Battery's charge capacity	Q	$60\ A \cdot h$
Initial state of charge	SOC_{init}	$86.9\ \%$
Full-state voltage	V_f	$399.8\ V$
Depleted-stated voltage	V_d	$343.22\ V$

5.2.3 Electric Converter

The *ElectricConverter* model implemented in the library represents an ideal DC/DC converter, whose behaviour is governed by the equation:

$$V_{Out} = D \cdot V_{In} \tag{5.12}$$

Where a positive Duty Cycle (D) indicates a power flow from the battery while a negative one indicates that it is the battery that is absorbing energy. In other words, the sign of D determines the power flow.

$$Power\ flow: \begin{cases} D < 0 & Charging\ Mode \\ D > 0 & Discharging\ Mode \\ D = 0 & Idle \end{cases} \tag{5.13}$$

This model disregards internal losses and transients, considering only a linear transfer of power between ports, with ideal conservation of energy:

$$Pw_{In} + Pw_{Out} = V_{In} \cdot I_{In} + V_{Out} \cdot I_{Out} = 0 \tag{5.12}$$

The purpose of this validation is to verify that the model responds adequately to dynamic variations in the duty cycle, and that the output voltage adjusts linearly to the expected value as a function of the input voltage.

To perform the experiment, a DC motor connected to a rotational load represented by an equivalent inertia has been used. The motor has been excited by means of an angular velocity reference signal composed of two components:

- A low frequency sinusoidal signal, which generates a continuous load variation on the converter.
- A PWM (pulse width modulation) signal, which introduces fast commutations in the duty cycle applied to the converter.

This combined stimulus allows to verify both the linearity of the model under smooth variations and its robustness against fast changes in the Duty Cycle.

Figure 5.5 shows the experimental scheme used in the validation. In it, it can be seen how the input signal is an angular velocity reference that feeds a PID block, in charge of regulating the performance of the DC/DC converter. This PID generates the DutyCycle signal that is applied to the converter, allowing to control the voltage supplied to the DC motor. Through this topology, it is possible to impose a desired speed evolution on the motor, while observing the system response to combined setpoint profiles (Square and Sine velocity reference).

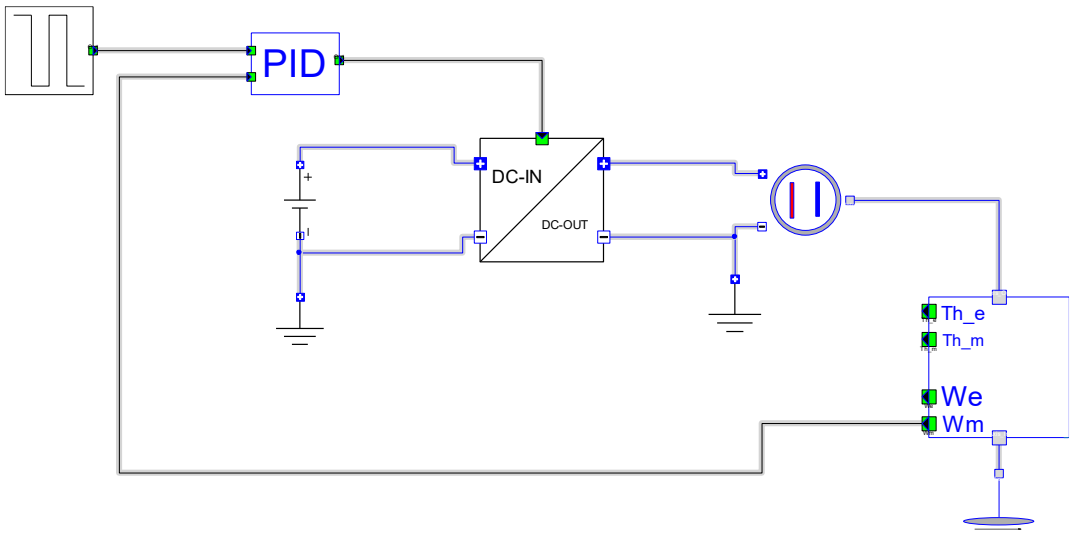

Figure 5.5: DC Converter Test Layout.

This test bench allows the converter to be subjected to controlled dynamic conditions, evaluating its behaviour against smooth and fast setpoint variations, as shown in Figures 5.6, 5.7. The results show an almost perfect match in all scenarios, which is to be expected in an idealized model. The energy balance always remains at 0 indicating that this model is neither a consumer nor a generator of energy to the system.

Table 5.4: DC Converter Validation Experiment Parameters.

Parameter	Symbol	Value
Motor armature resistance	R_m	0.025 Ω
Motor armature inductance	L_m	1 mH
Counter EMF constant	K_e	0.7144 $V \cdot s/rad$
Torque constant	K_t	0.72 $N \cdot m/A$
Rotor's friction constant	b_m	$5 \cdot 10^{-4}$ $N \cdot m \cdot s/rad$
Rotor's inertia	J	0.031 kg/m^2
DC Source Voltage	V	400 V
Load's inertia	J_{load}	10 kg/m^2
Load's Dynamic Viscosity	f_s	$8.5 \cdot 10^{-6}$ $Pa \cdot s$

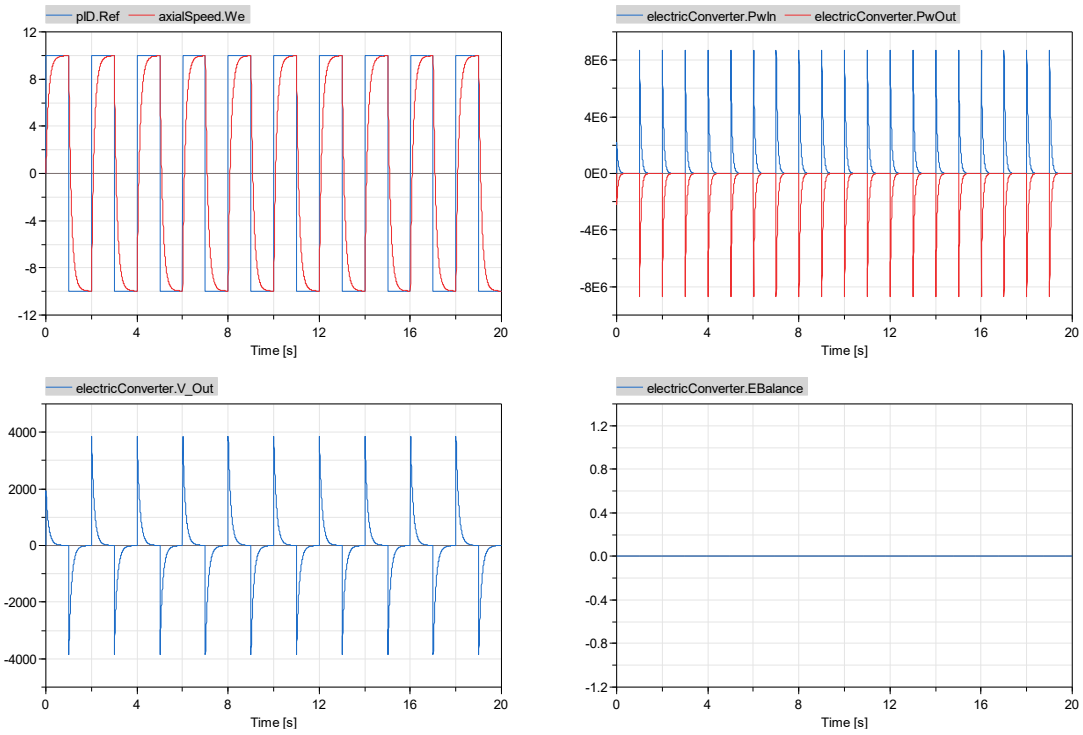

Figure 5.6: DC Converter and Motor Response to Square Velocity Reference.

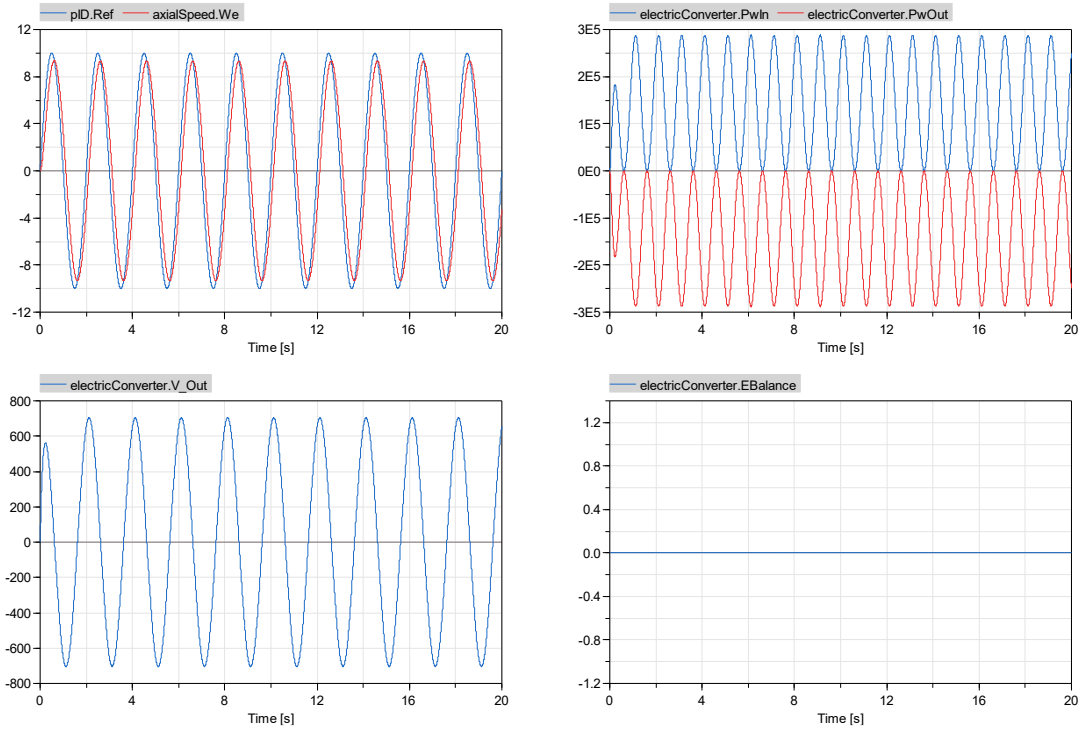

Figure 5.7: DC Converter and Motor Response to Sine Velocity Reference.

5.3 UDDS Cycle

To validate the behaviour of the proposed system and analyse its energy efficiency under realistic urban driving conditions, a model called *UDDS_Cycle* has been implemented in the EPowertrain library (Figure 5.8). This model reproduces the typical architecture of an electric power train and subjects it to a dynamic speed profile based on the standard UDDS (Urban Dynamometer Driving Schedule) cycle.

The *UDDS_Cycle* model integrates the following components:

- **Speed Profile Source (UDDS)**: A TimeTable block that defines the target vehicle speed as a function of time, following the UDDS cycle.
- **Power converter (ElectricConverter):** Implemented as an ideal voltage adapter to eliminate the need to represent commutations, which reduces numerical discontinuities and improves computational efficiency.
- **Motor (DCMotor):** Model previously validated using parametric identification techniques (Moments and Pasek), with parametrised resistance, inductance and torque constant values.
- **Chassis model (BodyFrame1DOF):** Represents the longitudinal dynamics of the vehicle and includes mass, ground friction, slope and aerodynamic drag.
- **Speed controller**: implemented through a PID control loop, the output controls the output voltage of the converter, and thus indirectly regulates the torque applied by the motor.

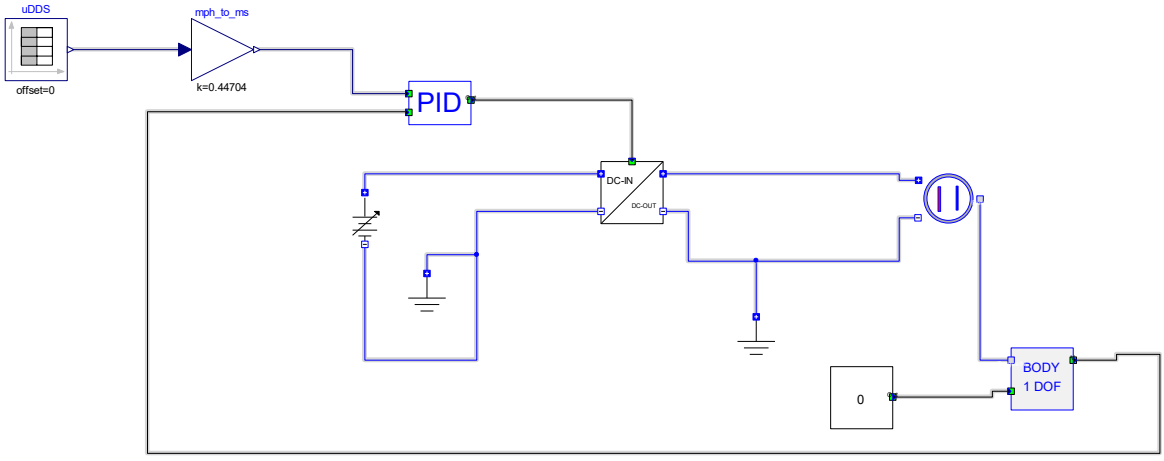

Figure 5.8: Test Layout of the UDDS Cycle Experiment.

Table 5.5: UDDS Experiment Parameters.

Parameter	Symbol	Value
Motor armature resistance	R_m	0.025 Ω
Motor armature inductance	L_m	1 mH
Counter EMF constant	K_e	0.65 $V \cdot s/rad$
Torque constant	K_t	0.65 $N \cdot m/A$
Rotor's friction constant	b_m	$4.21 \cdot 10^{-6}$ $N \cdot m \cdot s/rad$
Rotor's inertia	J	$3.87 \cdot 10^{-7}$ kg/m^2
Battery voltage (discharged)	V_d	500 V
Battery voltage (fully charged)	V_f	600 V
Battery capacity	Cap	30 $A \cdot h$
Initial SoC	$InitSOC$	0.8 (80%)
Battery resistance (series branch)	R_s	0.06 Ω
Battery resistance (parallel branch)	R_p	10^{-3} Ω
Battery capacitance	C	10^{-6} F
Battery maximum output current	I_{max}	400 A
Vehicle's mass	M	1500 kg
Vehicle's front area	A_f	2.2 m^2
Wheels radius	R	0.25 m
Vehicle's drag coefficient	C_d	0.29
Tyres' rolling friction coefficient	C_r	0.009
Air density	ρ	1.2 kg/m^3

Simulation Results
Figure 5.9 shows the results obtained by simulating the complete electric powertrain system under the standard UDDS (Urban Dynamometer Driving Schedule) cycle, with a duration of 1369 seconds.

Speed profile tracking
The controller achieves fully accurate tracking of the reference speed (pID.Ref) using the actual system signal (pID.In). This validates both the efficiency of the PID controller and the adequacy of the motor and mechanical system parameters.

Motor torque
The motor (dCMotor2.Rotor.T) shows torque variations from -900 Nm to +1000 Nm, with the following highlights:

- Positive values during acceleration phases.
- Negative peaks during regenerative braking.

These strong oscillations are to be expected in urban cycles [28], which highlights the dynamic demands of these scenarios.

Battery current
The output current (battery.Iout) shows:

- Peak consumption above 60 A.
- Negative currents (up to -40 A), indicating effective energy recovery.

The evolution in current is less violent than that of the motor torque due to the electrical inertia and the buffering capacity of the storage system.

State of charge (SOC)
The battery starts from a SOC of 80% and progressively drops to approximately 74.3%, reflecting a net energy consumption after compensation for regenerative recovery. This drop in SOC is within the expected range for an urban cycle and validates the size of the battery.

Energy balance and motor flux
electricConverter.EBalance reflects small variations around zero, which confirms a good power balance in the ideally modelled system. This is important to ensure that the ideal converter is not "creating" energy. The variations in this power balance are tiny and are due to the numerical deviation introduced by the finite precision of the solver.

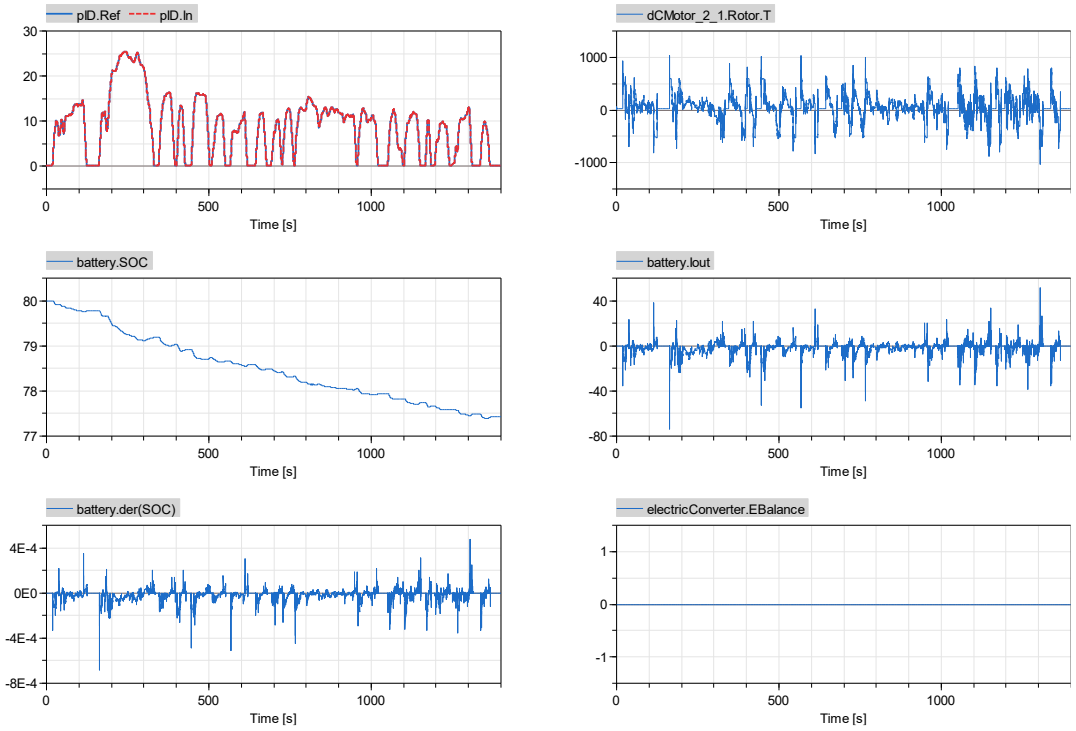

Figure 5.9: Results of the UDDS_Cycle Model. From Top to Bottom: Speed Tracking, Motor Torque, SOC, Battery Current, SoC Variation, Converter Energy Balance.

5.4 Real Driving Cycle Data

To validate the fidelity of the library, it is essential to check the simulation results against experimental data. For this purpose, driving data from the BWM i3 (60 Ah) have been used [2].

Model Identification

To test the simulation, we must first parameterise the model as accurately as possible. It is usually difficult to obtain all vehicle, battery and powertrain specific data, but we can approximate it.

Let us characterise the BWM i3 motor as an equivalent DC motor. For this we can rely on the technical data of the manufacturer [38]. According to the manufacturer the engine has a peak power of 125 kW with a nominal battery voltage of 360 V. This gives an approximate peak current of:

$$I_{peak} = \frac{P_{peak}}{V_{rated}} = \frac{125000W}{360V} = 347.22\ A \tag{6.1}$$

This is within the usual range of currents. From this maximum current we can estimate the torque constant:

$$k_T = \frac{\mathrm{T_{peak}}}{I_{peak}} = \frac{250Nm}{347.22A} = 0.72\ \frac{N \cdot m}{A} \tag{6.2}$$

The EMF constant can also be estimated. We know that the BWM i3 has a steady state power of 75 kW at 4800 rpm. Assuming a voltage V_{batt} of 360 V that gives a current of 208.33 A approx. We can calculate the counter-electromotive constant k_e as:

$$k_e = \frac{V_{EMF}}{\omega} \approx \frac{360\ V}{4800\ rpm} = \frac{360\ V}{502.65\ \frac{rad}{s}} = 0.7162\ \frac{V \cdot s}{rad} \tag{6.3}$$

Considering that not all the battery power is invested in overcoming the counter-electromotive force but that there are also losses in the power train, we will approximate k_e to k_t which is a common assumption for DC motors [39].

$$k_e \approx k_t \tag{6.4}$$

The internal resistance of the motor can also be derived from the power in steady state. considering that in steady state we can disregard the inductive component in a DC motor.

$$\frac{dI}{dt} \approx 0 \rightarrow \frac{P_{steady}}{V_{steady}} = \frac{75000\ W}{360\ V} = 208.33\ A \rightarrow R_m \approx \frac{360V}{208.33A} = 1.728\ \Omega \tag{6.5}$$

The inductance value has been estimated based on typical values for this type of motors.

$$L_{m_{est}} = 100\mu H \tag{6.6}$$

There are additional vehicle power losses to be considered. The most representative one is the energy expenditure in cabin cooling. In the case of [2], in the corresponding TripA01 cycle datasheet, the average power consumed by the air conditioning is:

$$P_{ac_{avg}} = 1610.4W \tag{6.7}$$

This additional power demand has been approximated by an additional charge to the battery:

$$R_{load} = \frac{P_{ac_{avg}}}{V_{batt}} = 80.4765\Omega \tag{6.8}$$

According to [38, 40, 41] , the rolling coefficient for C1 (passenger car) class tyres and C fuel efficiency on dry asphalt is:

$$0.0078 \leq C_r \leq 0.009 \tag{6.9}$$

On a first approach a middle value will be considered:

$$C_r = 0.0084 \tag{6.10}$$

The physical characteristics of the vehicle frame can be obtained directly from [38]. Although the specified gear ratio is 1:9.665, it has been modified to 1:6 to approximate the torque efforts recorded in the real data to those calculated for the speed profile. The complete parametrization of the experiment is shown below in Table 5.6.

Table 5.6: TripA01 Experiment Parametrization.

Parameter	Symbol	Value
TripA01 - Munich East	Drive cycle	[2]
Sunny	Weather	[2]
Motor armature resistance	R_m	1.72 Ω
Motor armature inductance	L_m	106.26 μH
Counter EMF constant	K_e	0.7144 $V \cdot s/rad$
Torque constant	K_t	0.72 $N \cdot m/A$
Rotor's friction constant	b_m	$5 \cdot 10^{-4}\ N \cdot m \cdot s/rad$
Rotor's inertia	J	$31 \cdot 10^{-3} kg/m^2$
Battery voltage (discharged)	V_d	343.22 V
Battery voltage (fully charged)	V_f	399.8 V
Battery capacity	Cap	60 $A \cdot h$
Initial SoC	$InitSOC$	86.9 %
Battery resistance (series branch)	R_s	0.1051 Ω
Battery resistance (parallel branch)	R_p	$1.7834 \cdot 10^{-4}\ \Omega$
Battery capacitance	C	$9.5329 \cdot 10^{3} F$
Battery maximum output current	I_{max}	400 A
Electric consumptions resistance	R_{load}	80.4765 Ω
Gear ratio	N	1:6*
Vehicle + driver mass	M	1280 kg
Vehicle's front area	A_f	2.38 m^2
Wheels' radius	R	0.29 m
Vehicle's drag coefficient	C_d	0.29
Tyres' rolling friction coefficient	C_r	0.0084
Air density	ρ	1.225 kg/m^3

5.5 Simulation Result

5.5.1 Variables Usually Compared

In the validation of electric vehicle models using real driving cycles, it is common to compare the main simulated vs. measured vehicle performance variables. Among the most common are:

- **Battery state of charge (SoC):** SoC over the cycle (or total SoC drop after a run) is a direct indicator of energy consumed. Validating the SoC evolution in the model against reality allows verifying whether the predicted energy consumption matches the real one. For example, Covello et al. compared the difference in SoC before and after each trip between simulation and experiment, as a measure of the energy consumed, obtaining very small discrepancies [42].

- **Battery current/power:** Battery current or instantaneous delivered/recuperated power is another key variable. Comparing the current profiles of the model with real data allows to assess whether the electrical system (battery, inverter, motor) is correctly reproducing the demands of the driving cycle. This also includes validating energy recovery (regen) under braking.

- **Vehicle speed**: Although typically the speed profile of the driving cycle is used as input, in models with a driver model controller it is verified that the simulation tracks the target speed accurately. The speed tracking error between the model and the actual cycle is analysed, especially in acceleration/braking transients. For example, Tollner et al. [43] define validation criteria for tracking the target speed and report that the largest deviations occur during hard braking or acceleration overshoots. A good model should minimise the speed error to ensure that the motor/battery load conditions are comparable to the real ones.

- **Engine torque**: When CAN or instrument data is available, the simulated engine torque (or wheel drive force) is often compared to the measured torque. This verifies that the propulsion model delivers the torque required by the cycle. In [43], measured vs. simulated engine torque was plotted during an urban cycle, and a good match was observed except at peaks where the drive controller introduced small overcorrections. The coincidence in torque (both in traction and regenerative braking) is fundamental to affirm that the longitudinal dynamics and the engine model are accurate.

In summary, the variables typically validated include speed (as a reference), power/torque delivered, battery current and SoC, and derived metrics such as energy consumed. **These direct comparisons between simulation and reality allow to evaluate the degree of realism of the model** in each key aspect of electric vehicle performance.

5.5.2 Accuracy Levels and Tolerable Deviations

The literature indicates that good quantitative agreement between model and experimental data is expected, although small deviations are always tolerated. In electric vehicle validation studies, relative errors in the order of one digit percentage for the main energy variables are often considered acceptable:

- **Deviation in energy consumption/SoC:** An error of less than ~5-10% in total consumed energy or SoC drop is usually considered reasonable. For example, Sandrini et al. [44] report that after validation with real data, the maximum differences were in the order of 5% in mechanical quantities (e.g. velocity, acceleration) and always below 10% in electrical quantities. Similarly, Tollner et al. [43] establish as a criterion that the relative difference in accumulated energy (consumed from the pack) must be less than 10% to consider the model validated. These ranges (5-10%) are a frequent threshold for judging the energy fidelity of the model.

- **Error in velocity tracking:** For the velocity profile, a tracking of the cycle is required. In simulation environments very narrow tolerances are imposed, e.g.

±1 mph (~1.6 km/h) instantaneous error, although in practice minor differences during transients are tolerated. What is important is that the model's average speed and acceleration distribution match the actual ones, so that the dynamic loading is equivalent [43].

- **Error in instantaneous electrical variables:** Simulated battery current and torque should follow the same profile as the actual data. It is common to calculate metrics such as root mean square error (RMSE) or root mean square error. Low values (a few percent of the range) indicate a good fit. In some cases, after calibration, almost imperceptible differences are obtained.

- **Deviation thresholds:** When errors exceed ~10% in energy or show systematic trends, they are usually interpreted as indicating that the model needs to be refined. Errors within ~5% are usually called good agreement. In fact, an informal consensus in vehicle simulation is that a variance of ~2% represents very high agreement, 5% is good, and above 10% requires justification or improvement of the model [44]. The uncertainty of the actual measurements must also be considered: for example, SoC sensors in vehicles can have tolerances of ±0.5% [42]. which puts a limit to the accuracy with which they can be expected.

In summary, a valid model should reproduce with high fidelity the trends of the measured variables, admitting only small discrepancies. Divergences of less than 5-10% are expected for global indicators (and ideally even smaller). Values within this range are considered acceptable and attributable to experimental uncertainty or reasonable simplifications. Larger discrepancies, on the other hand, require attention and explanation.

Deviations between simulation and reality are typically explained by measurement limits, different environmental conditions, model simplifications and control assumptions, or inherent variability. Academic work justifies that the observed discrepancies are within the expected range given these causes. If significant, improvements to the model are proposed or the effect of the cause is quantified. This critical analysis of deviations strengthens confidence in the model, showing that it is understood why and how much it differs from reality under certain circumstances.

5.5.3 Results Analysis

Figure 5.10 displays the results obtained in the validation of the model against real data. A good degree of agreement in the key variables analysed (SoC, battery current and motor torque) is achieved, which supports the model accuracy of the Modeling approach adopted. However, some localised deviations are observed in the SoC evolution and in the current profile during some phases of the real driving cycle.

On the one hand, part of the deviation observed in the SoC can be attributed to the fact that the model uses a manually adjusted gear ratio to better match the torque profile. This decision was made deliberately to compensate for the lack of accurate information about the actual vehicle's gearbox and to improve the tuning of the dynamic behaviour. Although it improves the torque response of the system, this simplification may slightly affect the accuracy of the current evolution and, consequently, of the SoC.

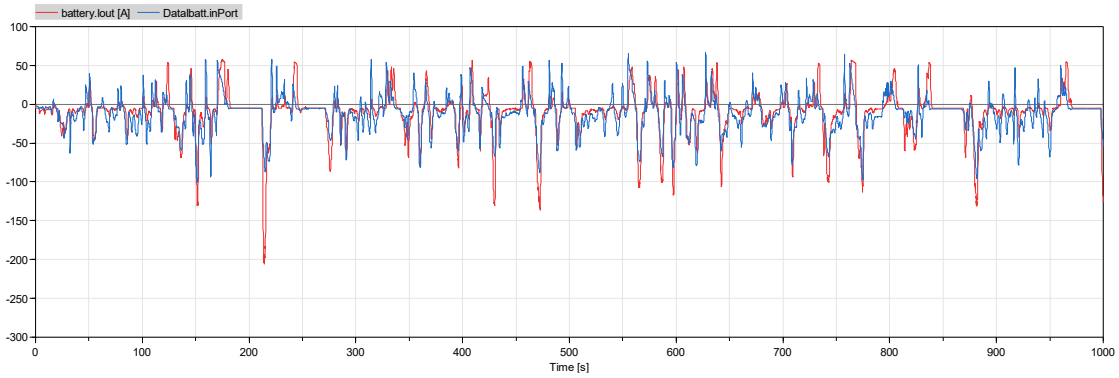

Figure 5.10: Battery Current Data vs Simulation.

Furthermore, it is important to consider that the experimental data used comes from a real campaign with a BMW i3 (60 Ah), where factors such as the activation of auxiliary systems (air conditioning, on-board electronics), ambient temperature, or the slope profile were not explicitly modelled in this version of the model. These external effects can introduce non-negligible variations in consumption, as documented in previous studies. For example, [42] found that real-world energy consumption is significantly affected by the use of auxiliary systems, especially in urban travel, leading to differences of more than 5% in the SoC (Figure 5.11) consumed for similar speed and distance profiles.

It should be noted that the error levels obtained (below 5 %) are in line with what is considered acceptable in the specialised literature. In fact, works such as [43] or [44] establish margins of 5-10 % as a reasonable threshold for the validation of electric vehicle models under real conditions. In this context, the observed discrepancies do not compromise the overall validity of the model and can be justified by the aforementioned factors: drivetrain simplifications, external conditions not modelled, and instrumentation limitations.

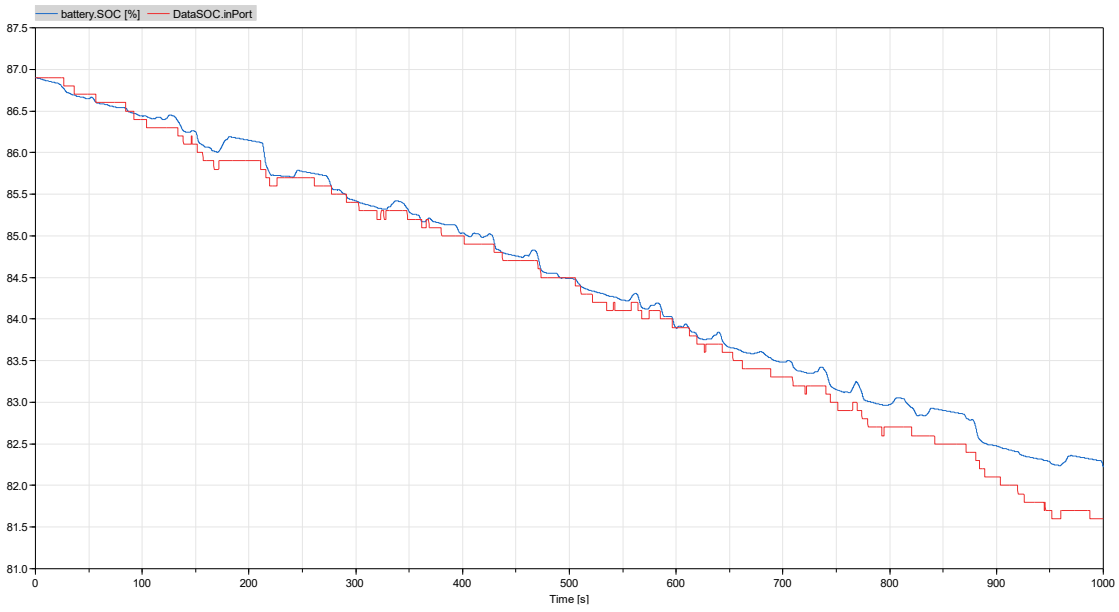

Figure 5.11: Simulated vs Data State of Charge.

On the other hand, the observed behaviour of the battery current and the simulated torque profile (Figure 5.12) , shows a good qualitative agreement with the real data. The point differences in the peaks can be explained by the idealised character of the driving controller employed, which does not fully replicate the decisions or smooth transitions of a human driver.

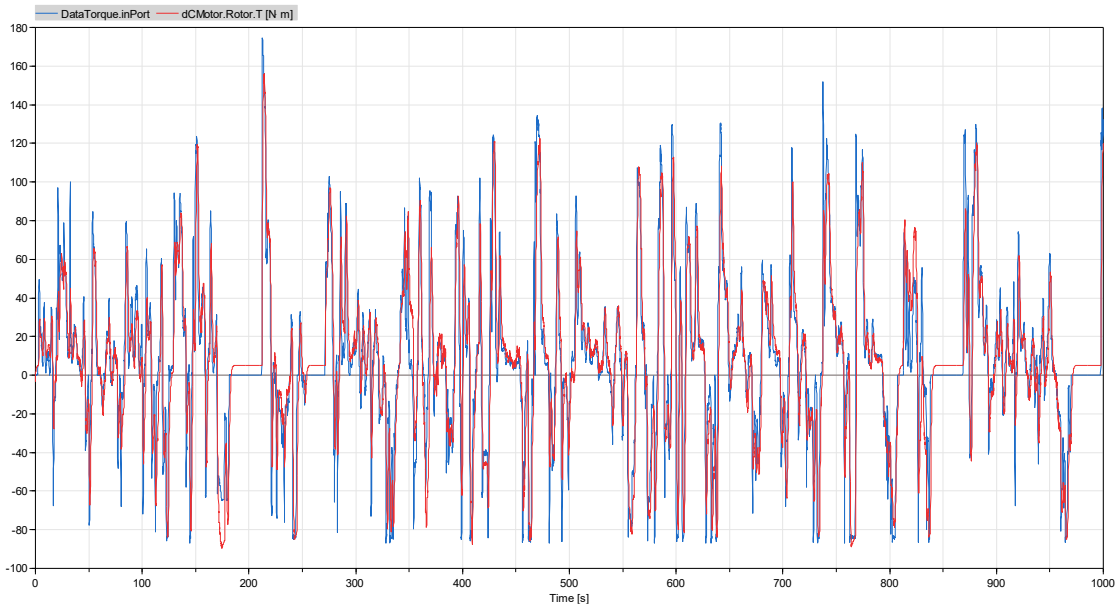

Figure 5.12: Simulated vs Data Torque.

In summary, these observations reinforce the validity of the model developed to simulate the energy behaviour of an electric vehicle in real conditions, while recognising its current limits and areas of improvement for future iterations. Figure 5.13 shows the main results of the experiment.

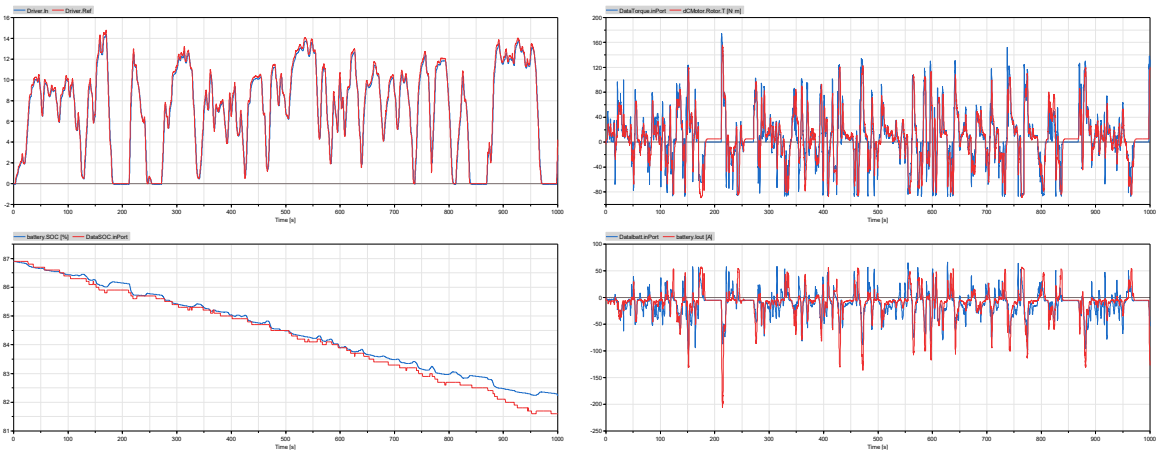

Figure 5.13: TripA01 Simulation Results.

5.6 Conclusions

The results obtained demonstrate that the models presented in this library, in spite of their simplicity, are capable of producing realistic results in dynamic and realistic environments.The observed deviations can be attributed to simplifications adopted in the modeling (e.g., absence of thermal modeling, idealized mechanical loads or ignored nonlinear saturations). Nevertheless, the overall results support the consistency of the approach followed and its usefulness as a modular, reproducible and extensible simulation platform.

Overall, the results provide support for the use of the library as a preliminary analysis tool in the design and evaluation of electric powertrains and set the basis for future extensions to increase its fidelity without compromising its operational efficiency.

Chapter 6 will then discuss how these results fit into the complete *EPowertrain* library project. As well as possible points of improvement.

6 Conclusions and Future Work

6.1 Conclusions

The completion of this project resulted in several important technical achievements, among which the following stand out:

- **Development of a modular library in Modelica:** A reusable and open architecture library was implemented in Modelica, capable of simulating the energetic behaviour of a complete electric powertrain for a passenger vehicle.

- **Configurable models of each subsystem:** The library integrates parametrizable models of the main components of the electric vehicle (lithium-ion battery, DC electric motor, power converter, control blocks and sensors), allowing the simulation to be easily adapted to different configurations and scenarios.

- **Validation with real data:** The performance of the complete model was validated by comparing the simulations with data from the standard UDDS urban driving cycle and with experimental data from a BMW i3. The results showed a good correlation between simulation and actual behaviour, demonstrating the accuracy and reliability of the developed models.

Overall, these achievements confirm that the objectives set for the thesis have been satisfactorily met. In addition to the technical aspects, the usefulness of the developed library is significant in both the educational and research fields.

In the educational context, this tool allows students and professionals to interactively explore the operation of an electric vehicle, examining how its different components interact in a safe simulation environment. The modular and transparent structure of the models facilitates learning, as it is possible to isolate subsystems (e.g. the battery or the motor) to study their individual behaviour and then understand their integration into the entire system. Also, the use of Modelica as an object-oriented, equation-based Modeling language provides a clear framework for assimilating concepts of physical Modeling and control of real systems.

From an academic perspective, the development process of this project has also entailed important methodological lessons. The adoption of an acausal Modeling approach, specific to the Modelica language, represented a relevant conceptual change with respect to traditional simulation tools based on causal models. This experience allowed us to appreciate the advantages of describing the system by means of declarative equations, without the need to predefine the direction of causation between variables, which results in greater flexibility and reusability of the models while maintaining the physical clarity of the internal relationships.

In addition, Modelica's object-oriented syntax facilitated the hierarchical organisation of the library and the encapsulation of components, reinforcing good practice in the design of complex models. Moreover, the multi-domain character of the approach (integrating

electrical, mechanical and control subsystems) provided a global view of the system and allowed simulating phenomena of different physical nature concurrently.

The latter was crucial for understanding the interdependencies in the powertrain: for example, how control decisions affect the current demand of the battery, or how battery limitations influence the torque delivery of the motor. In summary, the development of the library not only achieved the intended technical objectives but also provided a valuable training exercise in Modeling and simulation techniques for complex systems.

All these results and learnings demonstrate **the value of the project both for systems engineering education and for the advancement of electric vehicle Modeling**, providing the community with an effective tool and benchmark experience in the field of electric powertrain Modeling.

6.2 Future Work

The current work provided the basis for a modular and reusable library for electric powertrain simulation in Modelica. While the current implementation captures the main electromechanical dynamics and offers acceptable accuracy under real driving conditions, several lines of future development can be identified to extend its scope and fidelity:

- **Thermal domain integration:** one of the main simplifications of the current model is the absence of thermal Modeling. Including thermal sub-models for both the battery and the electrical machine would allow more accurate estimation of performance degradation under high load conditions, as well as the study of thermal protection strategies and thermal management systems (TMS). This is particularly relevant in contexts where phenomena such as regenerative braking or the use of auxiliary loads play a role, as reflected in the actual data used.

- **Advanced battery Modeling:** The current battery model is based on an equivalent lumped electrical circuit. Future extensions could incorporate electrochemistry-based models (e.g. Single Particle Model or Doyle-Fuller-Newman) or hybrid approaches to represent ageing phenomena, lithium deposition or temperature-dependent internal resistance. This would be particularly useful for long-term energy management studies.

- **System control improvements:** An ideal driver and a simplified voltage converter have been implemented in this version to reduce the computational burden. Including more realistic control logic - such as pedal mapping, regenerative braking thresholds or converter switching - would allow the impact of control strategies on energy consumption to be simulated more closely.

- **Inclusion of auxiliary systems:** The current model does not consider air conditioning systems and other auxiliary consumption, which in practice can represent a significant fraction of energy demand. Incorporating these elements would allow a more complete view of the vehicle's energy flow under varying conditions of environment and use.

- **Extension to other vehicle architectures:** The modular structure of the library facilitates its extension to hybrid (HEV), plug-in hybrid (PHEV) or fuel cell electric vehicles (FCEV). A future line of work could focus on the development of new components to simulate these architectures and compare their performance under equivalent driving cycles.

- **System-level optimisation and HIL compatibility:** Once validated, the library can serve as a basis for evaluating energy management strategies, component sizing or real-time optimisation. Its integration with hardware-in-the-loop (HIL) platforms would also enable rapid prototyping of controllers and embedded software.

- **Robust parameter identification:** Finally, the integration of data-driven tools for automatic parameter estimation from experimental records - especially for components such as the motor or battery - would help improve model accuracy and reduce the need for manual adjustments during validation.

These future lines of work would allow consolidating the proposed library as a flexible and robust tool for the analysis, design and optimisation of electric vehicles, contributing to the progress towards a more sustainable mobility.

Bibliography

[1] United States Enviromental Protection Agency (EPA), “Dynamometer Drive Schedules,” 18 04 2025. [Online]. Available: https://www.epa.gov/vehicle-and-fuel-emissions-testing/dynamometer-drive-schedules. [Accessed 27 05 2025].

[2] M. Steinstraeter, “ Battery and Heating Data in Real Driving Cycles,” ieee-dataport.org, 19 10 2020. [Online]. Available: https://ieee-dataport.org/open-access/battery-and-heating-data-real-driving-cycles. [Accessed 07 05 2025].

[3] K. Tammi, T. Minav and J. Kortelainen, “Thirty Years of Electro-Hybrid Powertrain Simulation,” *IEEE Access,* vol. 6, pp. 35250-35259, 2018.

[4] D. Zimmer, “Equation-Based Modeling with Modelica – Principles and Future Challenges.,” *SNE Simulation Notes Europe,* vol. 26, pp. 67-74, 06 2016.

[5] M. Ceraolo, “A new Modelica Electric and Hybrid Power Trains library,” *Proceedings of the 11th International Modelica Conference,* pp. 785-794, 09 2015.

[6] Modelica.org, “VehicleInterfaces Library,” 2 04 2025. [Online]. Available: https://github.com/modelica/VehicleInterfaces. [Accessed 22 05 2025].

[7] M. Dempsey, M. Gäfvert, P. Harman, C. Kral, M. Otter and P. Treffinger, “Coordinated automotive libraries for vehicle system Modeling,” in *5th International Modelica Conference*, Vienna, Austria, 2006.

[8] A. Romero and J. Angerer, “Fast Charge Algorithm Development for Battery Packs under Electrochemical and Thermal Constraints with JModelica.org,” in *Proceedings of the 15th International Modelica Conference*, Aachen, Germany, 2023.

[9] J. Batteh, M. Tiller, A. Goodman, Ford Motor Company, USA, “Monte Carlo Simulations for Evaluating Engine NVH Robustness,” in *Proceedings of the 4th International Modelica Conference*, Hamburg, 2005.

[10] The MathWorks Inc., “EV Reference Application,” he MathWorks, Inc., 2021. [Online]. Available: https://es.mathworks.com/help/autoblks/ug/electric-vehicle-reference-application.html. [Accessed 22 05 2025].

[11] E. Altuğ, Ö. Akyünci and E. Özgül, “Virtual battery electric vehicle development via 1D tools,” *Advances in Mechanical Engineering.,* vol. 15, 10 2023.

[12] D. Qin, J. Li, T. Wang and D. Zhang, “Modeling and Simulating a Battery for an Electric Vehicle Based,” *Automotive Innovation,* vol. 2, pp. 191-202, 08 2019.

[13] Modelica Association, “modelica.org,” [Online]. Available: https://modelica.org/. [Accessed 03 06 2025].

[14] W. Schamai, P. Fritzson, C. Paredis and A. Pop, “Towards Unified System Modeling and Simulation with ModelicaML Modeling of Executable Behavior Using Graphical Notations,” in *Proceedings of the 7th International Modelica Conference.*, Como, Italy, 2009.

[15] A. Haumer, “Modelica Standard Library,” modelica.org, 25 05 2025. [Online]. Available: https://github.com/modelica/ModelicaStandardLibrary. [Accessed 25 05 2025].

[16] T. M. N. Bui, T. Q. Dinh, J. Marco and C. Watts, “Development and Real-Time Performance Evaluation of Energy Management Strategy for a Dynamic Positioning Hybrid Electric Marine Vessel,” *Electronics,* vol. 10, 2021.

[17] R. Milishchuk and T. Bogodorova, “Thevenin-based Battery Model with Ageing Effects in Modelica,” in *IEEE 21st Mediterranean Electrotechnical Conference (MELECON)*, Palermo, Italy, 2022.

[18] LTX Simulation GmbH, “LTX Modelica Libraries Catalog,” 01 03 2019. [Online]. Available: https://www.ltx.de/download/Modelica_Libraries_Catalog_LTX.pdf. [Accessed 18 05 2025].

[19] M. Einhorn, F. V. Conte, C. Kral, C. Niklas, H. Popp and J. Fleig, “A Modelica Library for Simulation of Elecric Energy Storages,” in *Proceedings of the 8th International Modelica Conference*, Dresden, 2011.

[20] R. Yuan, T. Fletcher, A. Ahmedov, N. Kalantzis, A. Pezouvanis, N. Dutta, A. Watson and K. M. K. Ebrahimi, “Modeling and Co-simulation of hybrid vehicles: A thermal management perspective,” *Applied Thermal Engineering,* vol. 180, p. 115883, 08 2020.

[21] M. Chen and G. Rincon-Mora, “Accurate electrical battery model capable of predicting runtime and I-V performance,” *IEEE Transactions on Energy Conversion,* vol. 21, no. 2, pp. 504-511, 2006.

[22] C. Groß and A. W. Golubkov, “A Modelica library for Thermal-Runaway Propagation in Lithium-Ion Batteries,” in *Proceedings of 14th Modelica Conference*, Linköping, Sweden, 2021.

[23] A. Haumer, “Modelica.Thermal.HeatTransfer,” modelica.org, 26 08 2009. [Online]. Available: https://doc.modelica.org/om/Modelica.Thermal.HeatTransfer.html. [Accessed 17 05 2025].

[24] Politecnico di Milano, “ThermoPower Modelica Library,” 26 06 2024. [Online]. Available: https://build.openmodelica.org/Documentation/ThermoPower.html. [Accessed 17 05 2025].

[25] J. Marcicki, A. Conlisk and G. Rizzoni, “A lithium-ion battery model including electrical double layer effects,” *Journal of Power Sources,* vol. 251, pp. 157-169, 2014.

[26] J. Summerfield and C. Curtis, “Modeling the Lithium Ion/Electrode Battery Interface Using Fick’s Second Law of Diffusion, the Laplace Transform, Charge Transfer Functions, and a [4, 4] Padé Approximant,” *nternational Journal of Electrochemistry,* no. 10.1155/2015/496905, 2015.

[27] A. Innocenti, I. Álvarez, J.-F. Gohy and S. Passerini, “A modified Doyle-Fuller-Newman model enables the macroscale physical simulation of dual-ion batteries,” *Journal of Power Sources,* vol. 580, p. 233429, 1 10 2023.

[28] H. S. Ali, M. Hossein and F. M. Majid, “A new control algorithm of regenerative braking management for energy efficiency and safety enhancement of electric vehicles,” *Energy Conversion and Management,* vol. 276, p. 116564, 2023.

[29] F. D. Marco, “Comparative study of multiphysics Modeling and simulation software for lifetime performance evaluation of battery systems.,” Master’s Programme in Energy Storage – EIT InnoEnergy, 2023.

[30] M. Torabzadeh-Tari, P. Fritzson, M. Sjölund and A. Pop, “OpenModelica-Python Interoperability. Applied to Monte Carlo Simulation..,” in *Proceedings of the 50th Scandinavian Conference on Simulation and Modeling*, Fredericia, Denmark, 2009.

[31] J. Fang, M. Luo, J. Wang and Z. Hu, “FMI-Based Multi-Domain Simulation for an Aero-Engine Control System,” *Aerospace,* vol. 8, p. 180, 07 2021.

[32] ANSYS, “Ansys Twin Builder. Create and Deploy Digital Twin Models,” 2025. [Online]. Available: https://www.ansys.com/products/digital-twin/ansys-twin-builder#tab1-1. [Accessed 18 05 2025].

[33] B. Huang, Y. Hui, Y. Liu and H. Wang, “Design of Twin Builder-Based Digital Twin Online Monitoring System for Crane Girders,” *sensors,* vol. 23, no. 22, 2023.

[34] T. Ensbury, N. Horn and M. Dempsey, “Dymola and Simulink in Co-Simulation: A Vehicle Electronic Stability Control case study,” *American Modelica Conference,* 2020.

[35] dSPACE, “Integrating Functional Mock-up Units for HIL Simulation,” 2025. [Online]. Available: https://www.dspace.com/en/pub/home/news/event_modelica.cfm#179_16604. [Accessed 18 05 2025].

[36] M. Hadef and M. Mekideche, “Moments and Pasek’s methods for parameter identification of a DC motor,” *Journal of Zhejiang University-SCIENCE C (Computers & Electronics),* vol. 12, no. 2, pp. 124-131, 2011.

[37] C. Birkl and D. Howey, “Model identification and parameter estimation for LiFePO4 batteries,” in *IET Hybrid and Electric Vehicles Conference*, London, 2013.

[38] BMW, “autoblog.gr,” 07 2016. [Online]. Available: https://www.autoblog.gr/wp-content/uploads/2016/05/BMW_i3_Specifications_valid_0716.pdf. [Accessed 10 05 2025].

[39] IMC, Dipl.-Ing. Ingo Völlmecke, *Parameter Identification of DC Motors.*

[40] neumaticos-pneus-online.es, “Tyre 155/70 R19,” 11 05 2025. [Online]. Available: https://www.neumaticos-pneus-online.es/auto-neumatico-155-70-19.html. [Accessed 11 05 2025].

[41] European commision, “Regulation (EU) 2020/740 of the European Parliament and of the Council of 25 May 2020 on the labelling of tyres with respect to fuel efficiency and other parameters, amending Regulation (EU) 2017/1369 and repealing Regulation (EC) No 1222/2009,” 25 05 2020. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2020/740. [Accessed 11 05 2025].

[42] A. Covello, A. Di Martino and M. Longo, “Experimental Observation and Validation of EV Model for Real Driving Behavior,” *IEEE Access,* vol. 12, pp. 130763-130776, 2024.

[43] D. Tollner, Á. Nyerges, M. Jneid, A. Geleta and M. Zöldy, “How Do We Calibrate a Battery Electric Vehicle Model Based on Controller Area Network Bus Data?,” *Sensors,* vol. 24, 07 2024.

[44] G. Sandrini, M. Gadola and D. Chindamo, “Longitudinal Dynamics Simulation Tool for Hybrid APU and Full Electric Vehicle,” *Energies,* vol. 14, 2021.

[45] H. J., “Modeling of Hybrid Electric Vehicles in Modelica for Virtual Prototyping,” pp. 247-256, 2002.

[46] Z. S. K. Ahmed, “Power system simulation of fuel cell and supercapacitor based hybrid electric vehicle," International Journal of Hydrogen Energy,” vol. 40, no. 15, 2015.

[47] I. ANSYS, “Charge Up EV Development with Battery Digital Twins,” ANSYS, Inc, 2025. [Online]. Available: https://www.ansys.com/blog/charge-up-ev-development-with-battery-digital-twins. [Accessed 22 05 2025].

[48] L. Chang, J. Dai and S. Liu, “Design and feasibility analysis of a novel auto hold system in hydrostatic transmission wheeled vehicle,” *Automatika,* vol. 61, pp. 35-45, 2020.

[49] T.-S. Dao and C. Schmitke, “Developing Mathematical Models of Batteries in Modelica for Energy Storage Applications.,” in *The 11th International Modelica Conference*, Versailles, France, 2015.

[50] S. Chandra, A. C. Nair, A. K. Yadav and S. Singhal, “An integrated approach for Modeling Electric Powertrain,” in *5th Global Power, Energy and Communication Conference (GPECOM)*, Nevsehir, Turkiye, 2023.

Appendix A - The EPowertrain Modelica Library

The *EPowertrain* library has been developed at Modelica with the aim of offering a modular and flexible solution for the energy simulation of electric vehicles. Although it is contained in a single main package, its internal structure is organised in functional groups that group components with common purposes within the system architecture. This organisation favours reusability facilitates maintenance and improves the overall understanding of the model by the user or developer. The main functional groups included in the library are briefly described below:

- **Interfaces**: Includes models ports and connectors which allow coupling between electrical, mechanical and control components. Based on conservation of effort and flow, they are fundamental to maintain physical coherence in the system.

- **SignalRouting:** This package contains auxiliary components for the manipulation and distribution of internal signals within the model. It includes selectors, switches and conditional routing blocks that allow the implementation of discrete or decision logic.

- **Sources:** Gathers input signal sources such as speed, torque, voltage or current generators, used as stimuli to simulate specific operating conditions or standardised driving profiles.

- **Electrical**: Models such as *Resistance, Capacitor, IdealCoil, Diode, NMOS, PMOS, IGBT, IdealISwitch*. They represent basic elements of power electronics. This package also encompasses conversion components, such as DC-DC as well as electric machines for converting energy between electrical and mechanical domains.

- **Sensors**: Includes models such as *CurrentSensor, VoltageSensor,* or *AxialSpeed*. They are designed to measure key system variables without interfering with dynamic behaviour.

- **Mechanical**: Comprises models like *RotLoad, Wheel, BodyFrame1DOF, or TerrainSlope* which simulate the interaction between the powertrain and the vehicle's mechanical load, including terrain effects.

- **Control**: Includes components such as PID, PI, which allow the implementation and tuning of closed-loop control strategies for regulating power, speed, or other key variables.

- **Examples**: It contains complete configurations of electric vehicles under different operating conditions, integrating the above components to validate the li-

brary. These models allow driving cycles to be reproduced, energy consumption to be studied and control strategies to be compared.

The following sections present the most representative source code for each of these modules, with the original formatting and indentation preserved for readability and interpretation.

A1. Interfaces

```
package Interfaces
  connector PosPin
    Modelica.Units.SI.Voltage v;
    flow Modelica.Units.SI.Current i;
    annotation (Icon(graphics={
          Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,255},
            fillColor={0,0,255},
            fillPattern=FillPattern.Solid),
          Text(
            extent={{-98,-52},{100,-100}},
            textColor={255,255,255},
            textString=""),
          Rectangle(
            extent={{10,-60},{-10,60}},
            lineColor={28,108,200},
            fillColor={255,255,255},
            fillPattern=FillPattern.Solid,
            pattern=LinePattern.None),
          Rectangle(
            extent={{-60,10},{60,-10}},
            lineColor={28,108,200},
            fillColor={255,255,255},
            fillPattern=FillPattern.Solid,
            pattern=LinePattern.None)}));
  end PosPin;

  connector NegPin
    Modelica.Units.SI.Voltage v;
    flow Modelica.Units.SI.Current i;
    annotation (Icon(graphics={Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,255},
            fillColor={255,255,255},
            fillPattern=FillPattern.Solid), Rectangle(
            extent={{-60,10},{60,-10}},
            lineColor={28,108,200},
            fillColor={0,0,0},
            fillPattern=FillPattern.Solid,
            pattern=LinePattern.None)}));
  end NegPin;

  partial model ElectricPort

    SI.Voltage v "Voltage between pines (= p.u - n.u)";
    flow SI.Current i "Current from pin p to pin n";
```

```
public
  PosPin p annotation (Placement(transformation(extent={
            {-110,-10},{-90,10}}, rotation=0),
        iconTransformation(extent={{-110,-10},{-90,10}})));
  NegPin n annotation (Placement(transformation(extent={
            {70,-10},{90,10}}, rotation=0),
        iconTransformation(extent={{70,-10},{90,10}})));
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;

  annotation (Diagram(coordinateSystem(extent={{-100,-100},
            {100,100}})), Icon(coordinateSystem(extent={
            {-100,-100},{80,100}})));
end ElectricPort;

connector MechanicalAxis
  "Mechanical axis coupling"
  SI.Angle Phi;
  flow SI.Torque T;
  annotation (Icon(graphics={Rectangle(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,255},
          fillColor={215,215,215},
          fillPattern=FillPattern.Solid), Text(
          extent={{-98,-52},{100,-100}},
          textColor={255,255,255},
          textString="%name")}));
end MechanicalAxis;

connector IO_Port = Real annotation (Icon(graphics={
        Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,0},
            fillColor={0,255,0},
            fillPattern=FillPattern.Solid),Text(
            extent={{-44,138},{40,88}},
            textColor={0,0,0},
            textString="%name"),Polygon(
            points={{0,100},{100,0},{0,-100},{0,100}},
            lineColor={0,0,255},
            fillColor={0,0,0},
            fillPattern=FillPattern.Solid),Polygon(
            points={{0,100},{-100,0},{0,-100},{0,100}},
            lineColor={0,0,255})}));
connector BoolOutPort = output Boolean annotation (Icon(
      graphics={
      Rectangle(
        extent={{-100,100},{100,-100}},
        lineColor={0,0,0},
        fillColor={28,108,200},
        fillPattern=FillPattern.Solid),
      Text(
        extent={{-44,138},{40,88}},
        textColor={0,0,0},
        textString="%name"),
      Polygon(
        points={{0,100},{100,0},{0,-100},{0,100}},
        lineColor={0,0,255},
        fillColor={0,0,0},
```

```
        fillPattern=FillPattern.Solid)}));
connector BoolInPort = input Boolean annotation (Icon(
      graphics={Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,0},
            fillColor={28,108,200},
            fillPattern=FillPattern.Solid),Text(
            extent={{-44,138},{40,88}},
            textColor={0,0,0},
            textString="%name"),Polygon(
            points={{-100,100},{0,0},{-100,-100},{-100,100}},
            lineColor={0,0,255},
            fillColor={0,0,0},
            fillPattern=FillPattern.Solid)}));
connector OutPort = output Real annotation (Icon(
      graphics={
      Rectangle(
        extent={{-100,100},{100,-100}},
        lineColor={0,0,0},
        fillColor={0,255,0},
        fillPattern=FillPattern.Solid),
      Text(
        extent={{-44,138},{40,88}},
        textColor={0,0,0},
        textString="%name"),
      Polygon(
        points={{0,100},{100,0},{0,-100},{0,100}},
        lineColor={0,0,255},
        fillColor={0,0,0},
        fillPattern=FillPattern.Solid)}));
connector InPort = input Real annotation (Icon(graphics
      ={Rectangle(
        extent={{-100,100},{100,-100}},
        lineColor={0,0,0},
        fillColor={0,255,0},
        fillPattern=FillPattern.Solid), Polygon(
        points={{-100,100},{0,0},{-100,-100},{-100,100}},
        lineColor={0,0,255},
        fillColor={0,0,0},
        fillPattern=FillPattern.Solid)}));
partial model ThreePins

  EPowertrain.Interfaces.PosPin d "drain" annotation (
      Placement(transformation(extent={{-10,110},{10,90}},
          rotation=0)));
  EPowertrain.Interfaces.PosPin g "gate" annotation (
      Placement(transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  EPowertrain.Interfaces.PosPin s "source" annotation (
      Placement(transformation(extent={{-10,-110},{10,-90}},
          rotation=0)));

  annotation (Icon(graphics={Line(
              points={{0,90},{0,40}},
              color={0,0,0},
              thickness=1),Line(
              points={{0,40},{-20,40}},
              color={0,0,0},
              thickness=1),Line(
              points={{-20,40},{-20,-40}},
              color={0,0,0},
```

```
                thickness=1),Line(
                points={{-20,-40},{0,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,-40},{0,-90}},
                color={0,0,0},
                thickness=1),Line(
                points={{-30,40},{-30,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-90,0},{-46,0}},
                color={0,0,0},
                thickness=1)}));
end ThreePins;

model Voltage_to_Control
  PosPin posPin annotation (Placement(transformation(
          extent={{-120,-80},{-60,80}}, rotation=0)));
  OutPort outPort annotation (Placement(transformation(
          extent={{60,-80},{120,80}}, rotation=0)));
equation
  posPin.i = 0;
  posPin.v = outPort;
  annotation (
    Diagram(graphics),
    Icon(graphics={Line(
                points={{-90,0},{90,0}},
                color={0,0,0},
                thickness=0.5),Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={0,0,0},
                lineThickness=0.5)}),
    DymolaStoredErrors);
end Voltage_to_Control;

model Space3Phasor

  parameter Integer n=3 "Number of phases";
  constant Real pi=Modelica.Constants.pi;

  // Clark tranformation matrix
  Real T[2,n]=2/n*{{cos(k)/n*2*pi for k in 0:n - 1},{
      sin(k)/n*2*pi for k in 0:n - 1}};

  // Clark Inverse tranformation matrix
  Real InvT[n,2]={{cos(-k)/n*2*pi,-sin(-k)/n*2*pi} for
      k in 0:n - 1};

  Modelica.Units.SI.Voltage V[n]
    "instantaneous phase voltages";
  Modelica.Units.SI.Current I[n]
    "instantaneous phase currents";

  PosPin Vin[n] annotation (Placement(transformation(
          extent={{-110,-8},{-90,12}}, rotation=0)));
  PosPin Vout[2] annotation (Placement(transformation(
          extent={{90,50},{110,70}}, rotation=0)));
  NegPin Zero annotation (Placement(transformation(
          extent={{90,-50},{110,-30}}, rotation=0)));
equation
  V[:] = Vin[:].v;
```

```
    I[:] = Vin[:].i;
    Vout.v = T*V;
    Vout.i = -T*I;
    Zero.v = sum(V)/n;
    Zero.i = sum(I)/n;
    annotation (Diagram(graphics), Icon(graphics={
            Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={0,0,0}),Line(
                points={{-60,-40},{-60,78},{-60,80}},
                color={0,0,0}),Line(
                points={{-60,-40},{60,-40}},
                color={0,0,0}),Polygon(
                points={{-60,80},{-50,80},{-60,92},{-70,80},
              {-60,80}},
                lineColor={0,0,0}),Polygon(
                points={{60,-48},{70,-40},{60,-32},{60,-40},
              {60,-48}},
                lineColor={0,0,0}),Line(
                points={{-60,-40},{60,-80}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{-60,-40},{-20,80}},
                color={0,0,0},
                thickness=0.5),Polygon(
                points={{-24,78},{-18,76},{-20,80},{-24,78}},
                lineColor={0,0,0},
                lineThickness=0.5,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid),Polygon(
                points={{54,-82},{60,-80},{56,-76},{54,-82}},
                lineColor={0,0,0},
                lineThickness=0.5,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid)}));


  end Space3Phasor;

end Interfaces;
```

A2. SignalRouting

```
package SignalRouting
  model Not

    Interfaces.BoolInPort In annotation (Placement(
          transformation(extent={{-110,-10},{-90,10}},
            rotation=0)));
    Interfaces.BoolOutPort Out annotation (Placement(
          transformation(extent={{90,-10},{110,10}},
            rotation=0)));
  equation
    Out = not (In);
    annotation (Icon(graphics={Line(
                points={{-90,0},{-40,0},{-40,40},{20,0},{-40,
              -40},{-40,0}},
                color={0,0,0},
                thickness=0.5),Ellipse(
```

```
              extent={{20,6},{32,-6}},
              lineColor={0,0,0},
              lineThickness=0.5,
              fillColor={0,0,0},
              fillPattern=FillPattern.Solid),Line(
              points={{32,0},{90,0},{78,0}},
              color={0,0,0},
              thickness=0.5)}), Diagram(graphics));
end Not;

model Terminator

  Interfaces.InPort inPort annotation (Placement(
        transformation(extent={{-112,-10},{-92,10}},
          rotation=0)));
  annotation (Icon(graphics={Line(
              points={{-92,0},{20,0}},
              color={0,0,0},
              thickness=0.5),Line(
              points={{20,80},{20,-80},{-20,-80}},
              color={0,0,0},
              thickness=0.5),Line(
              points={{-20,80},{20,80}},
              color={0,0,0},
              thickness=0.5)}), Diagram(graphics));
end Terminator;

model unitDelay
  Interfaces.InPort inPort annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.OutPort outPort annotation (Placement(
        transformation(extent={{90,-12},{110,8}},
          rotation=0)));
initial equation
  outPort = 0;
equation
  when time > 0 then
    outPort = pre(inPort);
  end when;
  annotation (Icon(graphics={Text(
              extent={{20,60},{60,20}},
              textColor={0,0,0},
              textString="-1"),Text(
              extent={{-40,40},{40,-40}},
              textColor={0,0,0},
              textString="Z")}));

end unitDelay;

model Saturation

  Interfaces.InPort In annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.OutPort Out annotation (Placement(
        transformation(extent={{90,-12},{110,8}},
          rotation=0)));
  parameter Real Max=1e6;
  parameter Real Min=-1e-6;
equation
```

```
    Out = min(Max, max(Min, In));
    annotation (Icon(graphics={Line(
                points={{-80,80},{80,80}},
                color={0,0,255}),Line(
                points={{-80,-80},{80,-80},{80,-80}},
                color={0,0,255}),Line(
                points={{-80,-80},{-74,-80},{-60,-80},{-20,
              80},{14,80},{40,-76},{40,-80},{60,-80},{76,-6}},
                color={0,0,0},
                thickness=0.5)}), Diagram(graphics));

  end Saturation;

  model AND
    Interfaces.BoolInPort IN_1 annotation (Placement(
          transformation(extent={{-110,50},{-90,70}}),
          iconTransformation(extent={{-110,50},{-90,70}})));
    Interfaces.BoolInPort IN_2 annotation (Placement(
          transformation(extent={{-110,50},{-90,70}}),
          iconTransformation(extent={{-110,-70},{-90,-50}})));
    Interfaces.BoolOutPort Out annotation (Placement(
          transformation(extent={{88,-10},{108,10}}),
          iconTransformation(extent={{88,-10},{108,10}})));
  equation
    Out = if (IN_1 and IN_2) then true else false;
    annotation (Icon(coordinateSystem(preserveAspectRatio
            =false), graphics={Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={0,0,0},
                fillColor={255,255,255},
                fillPattern=FillPattern.Solid),Text(
                extent={{-98,56},{100,-142}},
                textColor={0,0,0},
                textString="AND
")}), Diagram(coordinateSystem(preserveAspectRatio=false)));

  end AND;

  package Mux "Multiplexers"
    model Mux_3_in

      Interfaces.InPort In3 annotation (Placement(
            transformation(extent={{-110,-70},{-90,-50}},
              rotation=0)));
      Interfaces.InPort In2 annotation (Placement(
            transformation(extent={{-110,-10},{-90,10}},
              rotation=0)));
      Interfaces.InPort In1 annotation (Placement(
            transformation(extent={{-110,50},{-90,70}},
              rotation=0)));
      Interfaces.OutPort out[3] annotation (Placement(
            transformation(extent={{90,-10},{110,10}},
              rotation=0)));
    equation
      out[1] = In1;
      out[2] = In2;
      out[3] = In3;
      annotation (Icon(graphics={Line(
                  points={{-90,60},{0,60},{0,0},{90,0}},
                  color={0,0,255}),Line(
                  points={{-90,0},{0,0}},
```

```
                color={0,0,255}),Line(
                points={{-90,-60},{0,-60},{0,0}},
                color={0,0,255})}));

end Mux_3_in;

model Mux_2_in

  Interfaces.InPort In2 annotation (Placement(
        transformation(extent={{-110,-70},{-90,-50}},
          rotation=0)));
  Interfaces.InPort In1 annotation (Placement(
        transformation(extent={{-110,50},{-90,70}},
          rotation=0)));
  Interfaces.OutPort out[2] annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
equation
  out[1] = In1;
  out[2] = In2;
  annotation (Icon(graphics={Line(
                points={{-90,60},{0,60},{0,0},{90,0}},
                color={0,0,255}),Line(
                points={{-90,-60},{0,-60},{0,0}},
                color={0,0,255}),Text(
                extent={{-40,58},{-80,98}},
                textColor={0,0,255},
                textString="1"),Text(
                extent={{-40,-62},{-80,-22}},
                textColor={0,0,255},
                textString="2")}), Diagram(graphics));

end Mux_2_in;

model Mux_2_in_Bool

  Interfaces.BoolInPort In2 annotation (Placement(
        transformation(extent={{-110,-70},{-90,-50}},
          rotation=0)));
  Interfaces.BoolInPort In1 annotation (Placement(
        transformation(extent={{-110,50},{-90,70}},
          rotation=0)));
  Interfaces.BoolOutPort out[2] annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
equation
  out[1] = In1;
  out[2] = In2;
  annotation (Icon(graphics={Line(
                points={{-90,60},{0,60},{0,0},{90,0}},
                color={0,0,255}),Line(
                points={{-90,-60},{0,-60},{0,0}},
                color={0,0,255}),Text(
                extent={{-40,58},{-80,98}},
                textColor={0,0,255},
                textString="1"),Text(
                extent={{-40,-62},{-80,-22}},
                textColor={0,0,255},
                textString="2")}), Diagram(graphics));

end Mux_2_in_Bool;
```

```
model ThreePhase_to_Bus

  Interfaces.NegPin In3 annotation (Placement(
        transformation(extent={{-110,-70},{-90,-50}},
          rotation=0)));
  Interfaces.NegPin In2 annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.NegPin In1 annotation (Placement(
        transformation(extent={{-110,50},{-90,70}},
          rotation=0)));
  Interfaces.PosPin out[3] annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
equation
  out[1] = In1;
  out[2] = In2;
  out[3] = In3;
  annotation (Icon(graphics={Line(
                points={{-90,60},{0,60},{0,0},{90,0}},
                color={0,0,255}),Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{-90,-60},{0,-60},{0,0}},
                color={0,0,255})}), Diagram(graphics));

end ThreePhase_to_Bus;

model Bus_to_ThreePhase

  Interfaces.PosPin out3 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
  Interfaces.PosPin out1 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.PosPin out2 annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
  Interfaces.NegPin In[3] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
equation
  In[1] = out1;
  In[2] = out2;
  In[3] = out3;
  annotation (Icon(graphics={Line(
                points={{-90,0},{86,0},{90,0}},
                color={0,0,255}),Line(
                points={{0,0},{0,60},{90,60}},
                color={0,0,255}),Line(
                points={{0,0},{0,-60}},
                color={0,0,255}),Line(
                points={{0,-60},{90,-60}},
                color={0,0,255})}), Diagram(graphics));

end Bus_to_ThreePhase;

model Demux_2
```

```
  Interfaces.InPort In[2] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.OutPort out1 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.OutPort out2 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
equation
  In[1] = out1;
  In[2] = out2;
  annotation (Icon(graphics={Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{90,-60},{0,-60},{0,0}},
                color={0,0,255}),Line(
                points={{0,0},{0,60},{90,60}},
                color={0,0,255}),Text(
                extent={{78,98},{40,60}},
                textColor={0,0,255},
                textString="1"),Text(
                extent={{80,-20},{42,-58}},
                textColor={0,0,255},
                textString="2")}), Diagram(graphics));

end Demux_2;

model Demux_2_Bool

  Interfaces.BoolInPort In[2] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.BoolOutPort out1 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.BoolOutPort out2 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
equation
  In[1] = out1;
  In[2] = out2;
  annotation (Icon(graphics={Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{90,-60},{0,-60},{0,0}},
                color={0,0,255}),Line(
                points={{0,0},{0,60},{90,60}},
                color={0,0,255}),Text(
                extent={{78,98},{40,60}},
                textColor={0,0,255},
                textString="1"),Text(
                extent={{80,-20},{42,-58}},
                textColor={0,0,255},
                textString="2")}), Diagram(graphics));

end Demux_2_Bool;

model Demux_3

  Interfaces.InPort In[3] annotation (Placement(
```

```
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.OutPort out1 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.OutPort out3 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
  Interfaces.OutPort out2 annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
equation
  In[1] = out1;
  In[2] = out2;
  In[3] = out3;
  annotation (Icon(graphics={Line(
                points={{90,60},{0,60},{0,0},{90,0}},
                color={0,0,255}),Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{90,-60},{0,-60},{0,0}},
                color={0,0,255}),Text(
                extent={{40,100},{80,60}},
                textColor={0,0,0},
                textString="1"),Text(
                extent={{40,40},{80,0}},
                textColor={0,0,0},
                textString="2"),Text(
                extent={{40,-20},{80,-60}},
                textColor={0,0,0},
                textString="3")}), Diagram(graphics));

end Demux_3;

model Demux_6

  Interfaces.InPort In[6] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.OutPort out1 annotation (Placement(
        transformation(extent={{90,90},{110,110}},
          rotation=0)));
  Interfaces.OutPort out3 annotation (Placement(
        transformation(extent={{90,10},{110,30}},
          rotation=0)));
  Interfaces.OutPort out2 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.OutPort out4 annotation (Placement(
        transformation(extent={{90,-30},{110,-10}},
          rotation=0)));
  Interfaces.OutPort out5 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
  Interfaces.OutPort out6 annotation (Placement(
        transformation(extent={{90,-110},{110,-90}},
          rotation=0)));
equation
  In = {out1,out2,out3,out4,out5,out6};
  annotation (Icon(coordinateSystem(
        preserveAspectRatio=false,
```

```
        preserveOrientation=false,
        extent={{-100,-120},{100,120}},
        initialScale=0.1), graphics={Line(
                points={{90,60},{0,60},{0,20},{90,20}},
                color={0,0,255}),Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{90,-60},{0,-60},{0,0}},
                color={0,0,255}),Line(
                points={{0,0},{0,100},{90,100}},
                color={0,0,255}),Line(
                points={{0,0},{0,-20},{90,-20}},
                color={0,0,255}),Line(
                points={{0,0},{0,-2},{0,-100},{92,-100}},
                color={0,0,255})}), Diagram(
        coordinateSystem(
        preserveAspectRatio=false,
        preserveOrientation=false,
        extent={{-100,-120},{100,120}},
        initialScale=0.1), graphics));

end Demux_6;

model BoolDemux_6

  Interfaces.BoolInPort In[6] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.BoolOutPort out1 annotation (Placement(
        transformation(extent={{90,90},{110,110}},
          rotation=0)));
  Interfaces.BoolOutPort out3 annotation (Placement(
        transformation(extent={{90,10},{110,30}},
          rotation=0)));
  Interfaces.BoolOutPort out2 annotation (Placement(
        transformation(extent={{90,50},{110,70}},
          rotation=0)));
  Interfaces.BoolOutPort out4 annotation (Placement(
        transformation(extent={{90,-30},{110,-10}},
          rotation=0)));
  Interfaces.BoolOutPort out5 annotation (Placement(
        transformation(extent={{90,-70},{110,-50}},
          rotation=0)));
  Interfaces.BoolOutPort out6 annotation (Placement(
        transformation(extent={{90,-110},{110,-90}},
          rotation=0)));
equation
  In = {out1,out2,out3,out4,out5,out6};
  annotation (Icon(coordinateSystem(
        preserveAspectRatio=false,
        preserveOrientation=false,
        extent={{-100,-120},{100,120}},
        initialScale=0.1), graphics={Line(
                points={{90,60},{0,60},{0,20},{90,20}},
                color={0,0,255}),Line(
                points={{-90,0},{0,0}},
                color={0,0,255}),Line(
                points={{90,-60},{0,-60},{0,0}},
                color={0,0,255}),Line(
                points={{0,0},{0,100},{90,100}},
                color={0,0,255}),Line(
```

```
                    points={{0,0},{0,-20},{90,-20}},
                    color={0,0,255}),Line(
                    points={{0,0},{0,-2},{0,-100},{92,-100}},
                    color={0,0,255})}), Diagram(
              coordinateSystem(
              preserveAspectRatio=false,
              preserveOrientation=false,
              extent={{-100,-120},{100,120}},
              initialScale=0.1), graphics));

    end BoolDemux_6;
  end Mux;

  model Multiply
    Interfaces.OutPort Out annotation (Placement(
          transformation(extent={{90,-10},{110,10}})));
    Interfaces.InPort In1 annotation (Placement(
          transformation(extent={{-110,30},{-90,50}})));
    Interfaces.InPort In2 annotation (Placement(
          transformation(extent={{-110,-52},{-90,-32}})));
  equation
    Out = In1*In2;
    annotation (Icon(coordinateSystem(preserveAspectRatio
          =false), graphics={Text(
              extent={{-80,60},{-20,20}},
              textColor={0,0,0},
              textStyle={TextStyle.Bold},
              textString="X"),Text(
              extent={{-80,-20},{-20,-60}},
              textColor={0,0,0},
              textStyle={TextStyle.Bold},
              textString="X"),Text(
              extent={{98,30},{38,-30}},
              textColor={0,0,0},
              textString=">"),Rectangle(
              extent={{-100,100},{98,-100}},
              lineColor={0,0,0},
              lineThickness=0.5)}), Diagram(
          coordinateSystem(preserveAspectRatio=false)));

  end Multiply;
end SignalRouting;
```

A3. Sources

```
package Sources
  import Modelica;
  model Ramp "Step DC voltage source"

    output Interfaces.OutPort Out annotation (Placement(
          transformation(extent={{90,-10},{110,10}},
            rotation=0)));


    parameter Modelica.Units.SI.Time St=1;
    parameter Real Slope(min=0);
    parameter Real InitV=0;
    parameter Real FinalV=1;
```

```
initial equation

  Out = InitV;

equation

  if time >= St then
    if (InitV <= FinalV and Out <= FinalV) then
      der(Out) = Slope;
    elseif (InitV > FinalV and Out >= FinalV) then
      der(Out) = -Slope;
    else
      der(Out) = 0;
    end if;
  else
    der(Out) = 0;
  end if;

  annotation (Icon(graphics={Rectangle(extent={{-100,100},
              {100,-100}}, lineColor={0,0,0}), Line(
          points={{-80,-80},{-20,-80},{20,80},{80,80}},
          color={0,0,0},
          thickness=1)}));
end Ramp;

model Constant

  parameter Real Value=0;
  EPowertrain.Interfaces.OutPort Out annotation (
      Placement(transformation(extent={{90,-8},{110,12}},
          rotation=0)));
equation
  Out = Value;
  annotation (Icon(graphics={Rectangle(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid), Text(
          extent={{-76,22},{72,-26}},
          textColor={0,0,0},
          textString="%Value")}));
end Constant;

model BoolConstant "Boolean constant"

  parameter Boolean Value=false;
  Interfaces.BoolOutPort Out annotation (Placement(
        transformation(extent={{90,-8},{110,12}},
          rotation=0)));
equation
  Out = Value;
  annotation (Icon(graphics={Rectangle(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid), Text(
          extent={{-76,22},{72,-26}},
          textColor={0,0,0},
          textString="%Value")}));
end BoolConstant;
```

```
model PWM "PWM voltage signal source"

  parameter Real D(
    min=0,
    max=1) = 0.5 "Duty cicle";
  parameter Real Max=1 "Maximun value";
  parameter Real Min=0 "Minimum value";
  parameter SI.Time time_start(min=0) = 0 "start time";
  parameter SI.Frequency f=1 "frequency";
  SI.Time t(start=0);
  SI.Time T=1/f;
  SI.Time TOn=D*T;

  constant Real K=1e5;

  Interfaces.OutPort Out annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
equation

  der(t) = 1;

  when (t >= T) then
    reinit(t, 0);
  end when;

  der(Out) = if (time >= time_start and t <= TOn) then
    K*(Max - Out) else K*(Min - Out);

  annotation (
    Icon(graphics={
        Line(
          points={{-40,80},{-60,80},{-60,80},{-60,80},{-80,
              80}},
          color={0,0,0},
          thickness=0.5),
        Line(
          points={{-40,80},{-40,-80},{-20,-80},{0,-80},{
              0,80},{40,80},{40,-80},{80,-80}},
          color={0,0,0},
          thickness=0.5),
        Rectangle(extent={{-100,100},{100,-100}},
            lineColor={0,0,0})}),
    DymolaStoredErrors,
    Diagram(graphics));
end PWM;

model Sine "AC Voltage source"

  parameter SI.Voltage U0=1 "Amplitude";
  parameter SI.Frequency f=50 "Frequency";
  parameter SI.Angle phi=0 "Phase shift";

protected
  parameter Modelica.Units.SI.AngularFrequency w=2*
      Modelica.Constants.pi*f;
public
  Interfaces.OutPort p annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
```

```
equation

  p = U0*sin(w*time + phi);
  annotation (Icon(graphics={Bitmap(extent={{-60,60},{60,
              -60}}, fileName="../UNED/TFM/f02nW.png")}),
      Diagram(graphics));
end Sine;

model Step "Step DC voltage source"

  output Interfaces.OutPort Out annotation (Placement(
        transformation(extent={{90,-10},{110,10}},
          rotation=0)));
  replaceable type T = Real;
  parameter Modelica.Units.SI.Time St=1;
  parameter T InitV=0;
  parameter T FinalV=1;

equation
  if time >= St then
    Out = FinalV;
  else
    Out = InitV;
  end if;

  annotation (Icon(graphics={Rectangle(extent={{-100,100},
              {100,-100}}, lineColor={0,0,0}), Line(
          points={{-78,-80},{0,-80},{0,80},{82,80}},
          color={0,0,0},
          thickness=1)}));
end Step;

// Define the enumeration for cycle files

// Model to select and use the cycle data based on the enum

model UDDS
  extends Modelica.Blocks.Sources.TimeTable(table=fill(0.0,
        0, 2));
initial equation

end UDDS;

model BoolStep "Boolean step  source"

  output Interfaces.BoolOutPort Out annotation (
      Placement(transformation(extent={{90,-10},{110,10}},
          rotation=0)));

  parameter Modelica.Units.SI.Time St=1;
  parameter Boolean InitV=false;
  parameter Boolean FinalV=true;

equation
  if time >= St then
    Out = FinalV;
  else
    Out = InitV;
  end if;

  annotation (Icon(graphics={Rectangle(extent={{-100,100},
```

```
                {100,-100}}, lineColor={0,0,0}), Line(
            points={{-78,-80},{0,-80},{0,80},{82,80}},
            color={0,0,0},
            thickness=1)}));
  end BoolStep;

  model BoolPWM "PWM voltage signal source"

    parameter Real D(
      min=0,
      max=1) = 0.5 "Duty cicle";
    parameter SI.Time time_start(min=0) = 0 "start time";
    parameter SI.Frequency f=1 "frequency";
    SI.Time t(start=0);
    SI.Time T=1/f;
    SI.Time TOn=D*T;

    Interfaces.BoolOutPort Out annotation (Placement(
          transformation(extent={{90,-10},{110,10}},
            rotation=0)));
  equation

    der(t) = 1;

    Out = (time >= time_start and t <= TOn);

    when (t >= T) then
      reinit(t, 0);
    end when;
    annotation (
      Icon(graphics={
          Line(
            points={{-40,80},{-60,80},{-60,80},{-60,80},{-80,
                80}},
            color={0,0,0},
            thickness=0.5),
          Line(
            points={{-40,80},{-40,-80},{-20,-80},{0,-80},{
                0,80},{40,80},{40,-80},{80,-80}},
            color={0,0,0},
            thickness=0.5),
          Rectangle(extent={{-100,100},{100,-100}},
              lineColor={0,0,0})}),
      DymolaStoredErrors,
      Diagram(graphics));
  end BoolPWM;
end Sources;
```

A4. Electrical

```
package Electrical

  package Sources
    model AC_Source "AC Voltage source"

      parameter SI.Voltage U0=1 "Amplitude";
      parameter SI.Frequency f=50 "Frequency";
      parameter SI.Angle phi=0 "Phase shift";
      SI.Angle theta;
```

```
  constant Real pi=Modelica.Constants.pi;

  SI.Voltage v;
  SI.Current i;


public
  EPowertrain.Interfaces.PosPin p annotation (
      Placement(transformation(extent={{-10,90},{10,110}},
          rotation=0)));
  EPowertrain.Interfaces.NegPin n annotation (
      Placement(transformation(extent={{-10,-110},{10,
           -90}}, rotation=0)));
initial equation
  theta = phi;

equation
  0 = p.i + n.i;
  i = p.i;
  v = p.v - n.v;

  when theta >= 2*pi then
    reinit(theta, 0);
  end when;

  der(theta) = 2*pi*f;

  v = U0*Modelica.Math.sin(theta);
  annotation (Icon(graphics={Bitmap(
                extent={{-60,60},{60,-60}},
                fileName="../UNED/TFM/f02nW.png"),
          Line( points={{0,-90},{0,-46}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{0,48},{0,90}},
                color={0,0,0},
                thickness=0.5),Text(
                extent={{50,10},{136,-12}},
                textColor={0,0,0},
                textString="%name")}), Diagram(
        graphics));
end AC_Source;

model DC_Source "DC voltage source"
  extends EPowertrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Voltage U0=12;
equation
  v = U0;
  annotation (Icon(graphics={Line(
                points={{-90,0},{-36,0},{-10,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{90,0},{10,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{-10,40},{-10,-40},{-10,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{10,20},{10,-20}},
                color={0,0,0},
```

```
                thickness=1),Line(
                points={{-80,20},{-60,20}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{-70,30},{-70,10}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{60,20},{80,20}},
                color={0,0,0},
                thickness=0.5)}));
end DC_Source;

model DC_Current_Source
  "Direct current source"
  extends EPowertrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Current I0=1;
equation
  i = I0;
  annotation (Icon(graphics={Ellipse(
                extent={{-60,60},{60,-60}},
                lineColor={0,0,0},
                lineThickness=1),Line(
                points={{-90,0},{-60,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{60,0},{90,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{-40,0},{40,0}},
                color={0,0,0},
                thickness=1),Polygon(
                points={{-12,0},{10,10},{10,-12},{-12,
            0}},lineColor={0,0,0},
                lineThickness=1,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid),
          Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={0,0,0})}));
end DC_Current_Source;

model VStep "Step DC voltage source"
  extends EPowertrain.Interfaces.ElectricPort
    annotation (Icon(Line(points=[-80,-80; 0,-80; 0,80;
            80,80], style(
          color=0,
          rgbcolor={0,0,0},
          thickness=8)), Rectangle(extent=[-100,100; 100,
            -100], style(
          color=0,
          rgbcolor={0,0,0},
          thickness=4))));

  Real v_step(start=v0);

  parameter Modelica.Units.SI.Time St=1;
  parameter Real v0=0;
  parameter Real vf=1;
equation
  v_step = if time < St then v0 else vf;
  v = v_step;
```

```
  annotation (Icon(graphics={Line(
                points={{80,80},{0,80},{0,-60},{0,-80},
            {-80,-80}},
                color={0,0,0},
                thickness=1),Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={0,0,0})}));
end VStep;

model Battery "DC voltage source"

  parameter Modelica.Units.NonSI.ElectricCharge_Ah
    Cap(min=0) = 1 "Battery capacity";

  parameter Modelica.Units.SI.Voltage Vd=0
    "Dischargued voltage (SOC = 0)";
  parameter Modelica.Units.SI.Voltage Vf=12
    "Full voltage (SOC = 100)";

  parameter Real InitSOC(
    quantity="Percent",
    final unit="1",
    final displayUnit="%",
    min=0,
    max=100) = 100 "Initial state of charge";

  Real SOC(
    quantity="Percent",
    final unit="1",
    final displayUnit="%",
    min=0,
    max=100);

  parameter SI.Resistance Rs=0.06 "Serie resistance";
  parameter SI.Resistance Rp=1e-3
    "Parallel resistance";
  parameter SI.Capacitance C=1e-6
    "Battery capacitance";
  parameter SI.Current Imax=400
    "Maximum, peak current";

  SI.Voltage Vout=posPin.v - negPin.v;
  SI.Current Iout(
    min=-Imax,
    max=Imax) = posPin.i;



  Interfaces.PosPin posPin annotation (Placement(
        transformation(extent={{-10,90},{10,110}},
          rotation=0)));
  Interfaces.NegPin negPin annotation (Placement(
        transformation(extent={{-10,-110},{10,-90}},
          rotation=0)));

  Basic.Resistance RP(R=Rp) annotation (Placement(
        transformation(
        origin={20,10},
        extent={{-10,-10},{10,10}},
        rotation=270)));
public
```

```
  Basic.Capacitor C1(C=C) annotation (Placement(
        transformation(
        origin={-22,10},
        extent={{-10,-10},{10,10}},
        rotation=270)));
  Basic.Resistance RS(R=Rs) annotation (Placement(
        transformation(
        origin={0,38},
        extent={{-10,-10},{10,10}},
        rotation=270)));
public
  Var_DC_Source Batt annotation (Placement(
        transformation(
        origin={0,-44},
        extent={{-10,-10},{10,10}},
        rotation=270)));
  Devices.CurrentSaturation currentSaturation(IMax=
        Imax, IMin=-Imax) annotation (Placement(
        transformation(
        extent={{-10,-10},{10,10}},
        rotation=90,
        origin={0,68})));
initial equation
  SOC = InitSOC;
  RS.v = 0;
  C1.v = 0;
equation
  der(SOC) = Iout/(3600*Cap);

  if SOC > 0 then
    Batt.v = Vd + (Vf - Vd)*SOC/100;
  else
    Batt.i = 0;
  end if;



  connect(C1.p, RS.n) annotation (Line(points={{-22,20},
          {-22,26},{-14,26},{-14,24},{0,24},{0,28}},
        color={0,0,255}));
  connect(RP.p, RS.n) annotation (Line(points={{20,20},
          {20,26},{14,26},{14,24},{0,24},{0,28}},
        color={0,0,255}));
  connect(C1.n, RP.n) annotation (Line(points={{-22,0},
          {-22,-4},{20,-4},{20,0}}, color={0,0,255}));
  connect(Batt.p, RP.n) annotation (Line(points={{0,-34},
          {0,-4},{20,-4},{20,0}}, color={0,0,255}));
  connect(Batt.n, negPin) annotation (Line(points={{0,
          -54},{0,-100}}, color={0,0,255}));
  connect(currentSaturation.In, RS.p) annotation (
      Line(points={{0,58},{0,48}}, color={0,0,255}));
  connect(currentSaturation.Out, posPin) annotation (
      Line(points={{0.2,78},{0,80},{0,100}}, color={0,
          0,255}));
  annotation (
    extent=[-30,0; -10,20],
    rotation=270,
    Icon(graphics={Line(
                points={{0,84},{0,22}},
                color={0,0,0},
                thickness=0.5),Line(
```

```
                points={{-40,20},{40,20}},
                color={0,0,0},
                thickness=1),Line(
                points={{-20,0},{20,0}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{-40,-20},{40,-20}},
                color={0,0,0},
                thickness=1),Line(
                points={{-20,-40},{20,-40}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{0,-40},{0,-90}},
                color={0,0,0},
                thickness=0.5),Line(
                points={{-40,-38},{46,40}},
                color={0,0,0},
                thickness=1),Polygon(
                points={{46,40},{32,40},{46,28},{46,40}},
                lineColor={0,0,0},
                lineThickness=1,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid)}),
    Placement(transformation(
        origin={-20,10},
        extent={{-10,-10},{10,10}},
        rotation=270)),
    extent=[-30,0; -10,20],
    rotation=270);


end Battery;

model Var_DC_Source
  "Variable DC voltage source"

  SI.Voltage v "Voltage between pines (= p.u - n.u)";
  flow SI.Current i "Current from pin p to pin n";

  Interfaces.PosPin p annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}}),
        iconTransformation(extent={{-110,-10},{-90,10}})));
  Interfaces.NegPin n annotation (Placement(
        transformation(extent={{90,-10},{110,10}}),
        iconTransformation(extent={{90,-10},{110,10}})));
equation

  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  annotation (Icon(graphics={Line(
                points={{-40,0},{14,0},{40,0}},
                color={0,0,0},
                thickness=1,
                origin={-50,0},
                rotation=360),Line(
                points={{40,0},{-40,0}},
                color={0,0,0},
                thickness=1,
                origin={50,0},
                rotation=360),Line(
```

```
                points={{0,40},{0,-40},{0,-40}},
                color={0,0,0},
                thickness=1,
                origin={-10,0},
                rotation=360),Line(
                points={{0,20},{0,-20}},
                color={0,0,0},
                thickness=1,
                origin={10,0},
                rotation=360),Line(
                points={{-10,0},{10,0}},
                color={0,0,0},
                thickness=0.5,
                origin={-70,20},
                rotation=360),Line(
                points={{0,10},{0,-10}},
                color={0,0,0},
                thickness=0.5,
                origin={-70,20},
                rotation=360),Line(
                points={{-10,0},{10,0}},
                color={0,0,0},
                thickness=0.5,
                origin={70,20},
                rotation=360),Line(
                points={{40,-40},{-40,40}},
                color={0,0,0},
                thickness=0.5,
                rotation=360),Polygon(
                points={{-7,7},{-1,-7},{7,3},{-7,7}},
                lineColor={0,0,0},
                lineThickness=0.5,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid,
                origin={-33,33},
                rotation=360)}));
end Var_DC_Source;

model Ground "Zero voltage reference"

  EPowertrain.Interfaces.PosPin p annotation (
      Placement(transformation(extent={{-10,110},{10,90}},
          rotation=0)));
equation
  p.v = 0;
  annotation (Diagram(graphics), Icon(graphics={Line(
                points={{0,90},{0,20}},
                color={0,0,0},
                thickness=1),Line(
                points={{-60,20},{60,20}},
                color={0,0,0},
                thickness=1),Line(
                points={{-40,0},{40,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{-20,-20},{20,-20}},
                color={0,0,0},
                thickness=1)}));
end Ground;

model Square "Squarevoltage signal source"
```

```
    Interfaces.PosPin Out annotation (Placement(
          transformation(extent={{90,-10},{110,10}},
            rotation=0)));

    parameter Real D(
      min=0,
      max=1) = 0.5 "Duty cicle";
    parameter SI.Time time_start(min=0) = 0
      "start time";
    parameter SI.Frequency f=1 "frequency";
    parameter SI.Voltage Von=1;
    parameter SI.Voltage Voff=0;
    parameter Real k=1000 "smooth factor";

    SI.Time t(start=0);
    SI.Time T=1/f;
    SI.Time eps=1e-6;
    SI.Time TOn=D*T;

  equation

    der(t) = 1;

    when (t >= T) then
      reinit(t, 0);
    end when;

    der(Out.v) = if (t < TOn) then k*(Von - Out.v)
       else k*(Voff - Out.v);


    annotation (
      Icon(graphics={Line(
                  points={{-40,80},{-60,80},{-60,80},{-60,
              80},{-80,80}},
                  color={0,0,0},
                  thickness=0.5),Line(
                  points={{-40,80},{-40,-80},{-20,-80},{
              0,-80},{0,80},{40,80},{40,-80},{80,-80}},
                  color={0,0,0},
                  thickness=0.5),Rectangle(
                  extent={{-100,100},{100,-100}},
                  lineColor={0,0,0})}),
      Diagram(graphics),
      experiment(
        StopTime=2,
        Tolerance=0.1,
        __Dymola_Algorithm="Dassl"));
  end Square;
end Sources;

package Basic
  model IdealCoil
    extends EPowertrain.Interfaces.ElectricPort;
    parameter Modelica.Units.SI.Inductance L=1
      "Inductance";
  equation
    v = smooth(1, L*der(i));

    annotation (Icon(graphics={Line(
```

```
                    points={{-90,0},{-64,0}},
                    color={0,0,0},
                    thickness=1),Line(
                    points={{64,0},{90,0}},
                    color={0,0,0},
                    thickness=1),Bitmap(
                    extent={{-60,60},{60,-60}},
                    fileName="../../Downloads/2560px-
Coil_illustration.svg.png")}))
          ;
      end IdealCoil;

      model Resistance "Ideal resistance"
        extends EPowertrain.Interfaces.ElectricPort;
        parameter Modelica.Units.SI.Resistance R=100
          "Resistance";
      equation
        v = smooth(1, R*i);
        annotation (Icon(graphics={Line(
                    points={{-90,0},{-60,0}},
                    color={0,0,0},
                    thickness=1),Line(
                    points={{-60,0},{-50,20},{-30,-20},{-10,
                20},{10,-20},{30,20},{50,-20}},
                    color={0,0,0},
                    thickness=1),Line(
                    points={{50,-20},{60,0},{90,0}},
                    color={0,0,0},
                    thickness=1)}), Diagram(graphics));
      end Resistance;

      model VariableResistance
        "Variable parameter resistance"
        extends EPowertrain.Interfaces.ElectricPort;
        Modelica.Units.SI.Resistance R "Resistance";
      equation
        v = smooth(1, R*i);
        annotation (Icon(graphics={Line(
                    points={{-90,0},{-60,0}},
                    color={0,0,0},
                    thickness=1),Line(
                    points={{-60,0},{-50,20},{-30,-20},{-10,
                20},{10,-20},{30,20},{50,-20}},
                    color={0,0,0},
                    thickness=1),Line(
                    points={{50,-20},{60,0},{90,0}},
                    color={0,0,0},
                    thickness=1)}), Diagram(graphics));
      end VariableResistance;

      model Capacitor "Ideal capacitor"
        extends EPowertrain.Interfaces.ElectricPort;
        parameter Modelica.Units.SI.Capacitance C=1e-6
          "Capacitance";
      equation
        i = smooth(1, C*der(v));

        annotation (Diagram(graphics={Line(
                    points={{-20,40},{-20,-40}},
                    color={0,0,0},
                    thickness=1),Line(
```

```
                points={{20,40},{20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-88,0},{-20,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,0},{88,0}},
                color={0,0,0},
                thickness=1)}), Icon(graphics={Line(
                points={{-20,40},{-20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,40},{20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-88,0},{-20,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,0},{88,0}},
                color={0,0,0},
                thickness=1)}));
end Capacitor;

model VariableCapacitor
  "Variable parameter capacitor"
  extends EPowertrain.Interfaces.ElectricPort;
  Modelica.Units.SI.Capacitance C "Capacitance";
equation
  i = smooth(1, C*der(v));

  annotation (Diagram(graphics={Line(
                points={{-20,40},{-20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,40},{20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-88,0},{-20,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,0},{88,0}},
                color={0,0,0},
                thickness=1)}), Icon(graphics={Line(
                points={{-20,40},{-20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,40},{20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-88,0},{-20,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{20,0},{88,0}},
                color={0,0,0},
                thickness=1)}));
end VariableCapacitor;

model Y_Conection

  parameter Modelica.Units.SI.Capacitance C=1;
```

```
  Capacitor capacitor(C=C) annotation (Placement(
        transformation(extent={{48,-10},{68,10}})));
  Capacitor capacitor1(C=C) annotation (Placement(
        transformation(extent={{46,16},{70,40}})));
  Capacitor capacitor2(C=C) annotation (Placement(
        transformation(extent={{46,-38},{66,-18}})));
  Sources.Ground ground annotation (Placement(
        transformation(extent={{84,-20},{104,0}})));
  Interfaces.PosPin posPin[3] annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}})));
  SignalRouting.Mux.Bus_to_ThreePhase
    bus_to_ThreePhase annotation (Placement(
        transformation(extent={{-82,-46},{10,46}})));
equation
  connect(capacitor1.n, capacitor.n) annotation (Line(
        points={{70,28},{70,0},{68,0}}, color={0,0,255}));
  connect(capacitor2.n, capacitor.n) annotation (Line(
        points={{66,-28},{70,-28},{70,0},{68,0}},
        color={0,0,255}));
  connect(bus_to_ThreePhase.out2, capacitor.p)
    annotation (Line(points={{10,0},{48,0}}, color={0,
          0,255}));
  connect(bus_to_ThreePhase.out3, capacitor2.p)
    annotation (Line(points={{10,-27.6},{42,-27.6},{42,
          -28},{46,-28}}, color={0,0,255}));
  connect(bus_to_ThreePhase.out1, capacitor1.p)
    annotation (Line(points={{10,27.6},{42,27.6},{42,28},
          {46,28}}, color={0,0,255}));
  connect(capacitor.n, ground.p) annotation (Line(
        points={{68,0},{70,0},{70,6},{94,6},{94,0}},
        color={0,0,255}));
  connect(posPin, bus_to_ThreePhase.In) annotation (
      Line(points={{-100,0},{-82,0}}, color={0,0,255}));
  annotation (Icon(coordinateSystem(
          preserveAspectRatio=false), graphics={Line(
                points={{-70,-70},{0,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,0},{70,-70}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,0},{0,80}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,0},{0,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-20,-40},{20,-40}},
                color={0,0,0},
                thickness=1),Line(
                points={{-10,-50},{10,-50}},
                color={0,0,0},
                thickness=1),Line(
                points={{-4,-60},{4,-60}},
                color={0,0,0},
                thickness=1)}), Diagram(
        coordinateSystem(preserveAspectRatio=false)));
end Y_Conection;

model Delta_Conection
```

```
    parameter Modelica.Units.SI.Capacitance C=1e-6;

    Capacitor C_bc(C=C) annotation (Placement(
          transformation(
          extent={{-7,-7.99998},{7,8}},
          rotation=270,
          origin={40,-11})));
    Interfaces.PosPin posPin[3] annotation (Placement(
          transformation(extent={{-110,-10},{-90,10}})));
    SignalRouting.Mux.Bus_to_ThreePhase
      bus_to_ThreePhase annotation (Placement(
          transformation(extent={{-60,-40},{20,40}})));
    Capacitor C_ab(C=C) annotation (Placement(
          transformation(
          extent={{-7,-7.99999},{7,7.99999}},
          rotation=270,
          origin={40,11})));
    Capacitor C_ac(C=C) annotation (Placement(
          transformation(
          extent={{-7,-7.99998},{7,8}},
          rotation=270,
          origin={60,1})));
  equation
    connect(bus_to_ThreePhase.out3, C_bc.n) annotation
      (Line(points={{20,-24},{40,-24},{40,-18}}, color={
            0,0,255}));
    connect(bus_to_ThreePhase.out1, C_ab.p) annotation
      (Line(points={{20,24},{40,24},{40,18}}, color={0,0,
            255}));
    connect(C_ac.p, C_ab.p) annotation (Line(points={{60,
            8},{60,24},{40,24},{40,18}}, color={0,0,255}));
    connect(C_ac.n, C_bc.n) annotation (Line(points={{60,
            -6},{60,-24},{40,-24},{40,-18}}, color={0,0,
            255}));
    connect(bus_to_ThreePhase.out2, C_bc.p) annotation
      (Line(points={{20,0},{40,0},{40,-4}}, color={0,0,255}));
    connect(C_ab.n, C_bc.p) annotation (Line(points={{40,
            4},{40,0},{40,-4},{40,-4}}, color={0,0,255}));
    connect(posPin, bus_to_ThreePhase.In) annotation (
        Line(points={{-100,0},{-60,0}}, color={0,0,255}));
    annotation (Icon(coordinateSystem(
            preserveAspectRatio=false), graphics={
            Polygon(
                  points={{-80,-80},{0,80},{80,-80},{-80,
              -80}},
                  lineColor={0,0,0},
                  lineThickness=1),Rectangle(
                  extent={{-100,100},{100,-100}},
                  lineColor={0,0,0})}), Diagram(
          coordinateSystem(preserveAspectRatio=false)));
  end Delta_Conection;
end Basic;

package Semiconductors
  model NMOS "Ideal NMOS"

    EPowertrain.Interfaces.PosPin d "drain" annotation (
        Placement(transformation(extent={{-10,110},{10,90}},
            rotation=0)));
    EPowertrain.Interfaces.PosPin g "gate" annotation (
        Placement(transformation(extent={{-110,-10},{-90,
```

```
            10}}, rotation=0)));
  EPowertrain.Interfaces.PosPin s "source" annotation
    (Placement(transformation(extent={{-10,-110},{10,-90}},
          rotation=0)));

  Modelica.Units.SI.Current Ids(start=0);
  Modelica.Units.SI.Voltage Vgs(start=0)
    "Pins voltage (= g.v - s.v)";
  Modelica.Units.SI.Voltage Vds(start=0)
    "Pins voltage  (= d.v - s.v)";

  parameter Modelica.Units.SI.Voltage Vt=3;
  parameter Modelica.Units.SI.Length L=2e-6;
  parameter Modelica.Units.SI.Length W=10e-6;
  parameter Real Kp(unit="A/V^2") = 100e-6;
  parameter Real Lambda(unit="V^-1") = 0;

equation
  Vgs = g.v - s.v;
  Vds = d.v - s.v;

  g.i = 0;
  d.i = Ids;
  d.i = -s.i;

  // Cut
  if (Vgs < Vt) then
    Ids = 0;
  else
    // Lineal
    if (Vds <= (Vgs - Vt)) then
      Ids = Kp*(W/L)*((Vgs - Vt)*Vds - (Vds^2)/2)*(1 +
        Lambda*Vds);
      // Saturation
    else
      Ids = 0.5*Kp*(W/L)*(Vgs - Vt)^2*(1 + Lambda*Vds);
    end if;
  end if;

  annotation (Icon(graphics={Line(
                points={{-46,0},{-30,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{-90,0},{-46,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,90},{0,40},{-20,40},{-20,-40},
            {0,-40},{0,-90}},
                color={0,0,0},
                thickness=1),Line(
                points={{-30,40},{-30,-40}},
                color={0,0,0},
                thickness=1),Polygon(
                points={{0,-40},{-10,-34},{-10,-46},{0,
            -40}},
                lineColor={0,0,0},
                lineThickness=1,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid)}),
      DymolaStoredErrors);
end NMOS;
```

```
model PMOS "Ideal PMOS"

  EPowertrain.Interfaces.PosPin d "source" annotation
    (Placement(transformation(extent={{-10,-110},{10,-90}},
          rotation=0)));
  EPowertrain.Interfaces.PosPin g "gate" annotation (
      Placement(transformation(extent={{-110,-10},{-90,
            10}}, rotation=0)));
  EPowertrain.Interfaces.PosPin s "drain" annotation (
      Placement(transformation(extent={{-10,110},{10,90}},
          rotation=0)));

  Modelica.Units.SI.Current Ids(start=0);
  Modelica.Units.SI.Voltage Vgs(start=0)
    "Pins voltage (= g.v - s.v)";
  Modelica.Units.SI.Voltage Vds(start=0)
    "Pins voltage (= d.v - s.v)";

  parameter Modelica.Units.SI.Voltage Vt=3;
  parameter Modelica.Units.SI.Length L=2e-6;
  parameter Modelica.Units.SI.Length W=10e-6;
  parameter Real Kp(unit="A/V^2") = 100e-6;
  parameter Real Lambda(unit="V^-1") = 0;

equation
  Vgs = g.v - s.v;
  Vds = d.v - s.v;

  g.i = 0;
  d.i = Ids;
  d.i = -s.i;

  // Cut
  if (Vgs > Vt) then
    Ids = 0;
  else
    // Lineal
    if (Vds >= (Vgs - Vt)) then
      Ids = Kp*(W/L)*((Vgs - Vt)*Vds - (Vds^2)/2)*(1 +
        Lambda*Vds);
      // Saturation
    else
      Ids = 0.5*Kp*(W/L)*(Vgs - Vt)^2*(1 + Lambda*Vds);
    end if;
  end if;

  annotation (Icon(graphics={Ellipse(
                extent={{-46,8},{-30,-8}},
                lineColor={0,0,0},
                lineThickness=1),Line(
                points={{-90,0},{-46,0},{-48,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{0,90},{0,40},{-20,40},{-20,-40},
            {0,-40},{0,-90}},
                color={0,0,0},
                thickness=1),Line(
                points={{-30,40},{-30,-40}},
                color={0,0,0},
                thickness=1),Polygon(
```

```
                points={{-20,40},{-10,46},{-10,34},{-20,
            40}},
                lineColor={0,0,0},
                lineThickness=1,
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid)}));
end PMOS;

model Diode "Ideal diode"
  extends EPowertrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Current Is=1e-6
    "Saturation current";
  parameter Modelica.Units.SI.Voltage Vt=0.04
    "Thermal voltage";
  parameter Real Maxexp(final min=Modelica.Constants.small)
     = 15 "Max. exponent for linear continuation";
  parameter SI.Resistance R=1.e8
    "Parallel ohmic resistance";
equation

  i = smooth(1, (if (v/Vt > Maxexp) then Is*(exp(
    Maxexp)*(1 + v/Vt - Maxexp) - 1) + v/R else Is*(
    exp(v/Vt) - 1) + v/R));
  annotation (Icon(graphics={Line(
                points={{-84,0},{86,0}},
                color={0,0,0},
                thickness=1),Polygon(
                points={{0,0},{-20,20},{-20,-20},{0,0}},
                lineColor={0,0,0},
                lineThickness=1),Line(
                points={{0,20},{0,-20}},
                color={0,0,0},
                thickness=1)}), DymolaStoredErrors);
end Diode;

model IdealIGBT
  extends EPowertrain.Interfaces.ElectricPort;
  parameter SI.Current Is=1e-9 "Saturation current";
  parameter SI.Voltage Vt=0.025 "Thermal voltage";
  parameter Real Maxexp(final min=Modelica.Constants.small)
     = 15 "Max. exponent for linear continuation";
  parameter SI.Resistance R=1.e8
    "Parallel ohmic resistance";
  EPowertrain.Interfaces.PosPin c annotation (
      Placement(transformation(extent={{-20,-100},{0,-80}},
          rotation=0)));
equation
  c.i = 0;
  if (c.v > 0) then
    i = smooth(1, (if (v/Vt > Maxexp) then Is*(exp(
      Maxexp)*(1 + v/Vt - Maxexp) - 1) + v/R else Is*(
      exp(v/Vt) - 1) + v/R));
  else
    i = smooth(1, v/R);
  end if;

  annotation (Icon(graphics={Line(
                points={{-84,0},{86,0}},
                color={0,0,0},
                thickness=1),Polygon(
                points={{0,0},{-20,20},{-20,-20},{0,0}},
```

```
                lineColor={0,0,0},
                lineThickness=1),Line(
                points={{0,20},{0,-20}},
                color={0,0,0},
                thickness=1),Ellipse(
                extent={{-34,24},{14,-24}},
                lineColor={0,0,0},
                lineThickness=0.5),Line(
                points={{-10,-80},{-10,-76},{-10,-26}},
                color={0,0,0},
                pattern=LinePattern.Dash,
                thickness=0.5)}), Diagram(graphics));
end IdealIGBT;

model IdealISwitch
  extends EPowertrain.Interfaces.ElectricPort;
  parameter SI.Resistance ROpen=1e5
    "Opened circuit conductance";
  parameter SI.Resistance RClose=1.e-5
    "Closed circuit resistance";
  parameter SI.Time St(min=1e-9) = 1e-6 "Switch time";

  Real Smooth_aux
    "Aux variable to smooth resistance variation";

  SI.Resistance R(start=ROpen);



  Interfaces.BoolInPort c annotation (Placement(
        transformation(extent={{-20,-100},{0,-80}},
          rotation=0)));


equation

  der(Smooth_aux)*St = if c then 1 - Smooth_aux else -
    Smooth_aux;

  R = ROpen + (RClose - ROpen)*Smooth_aux;

  i = v/R;


  annotation (
    Icon(graphics={Line(
                points={{-10,-80},{-10,-76},{-10,-26}},
                color={0,0,0},
                pattern=LinePattern.Dash,
                thickness=0.5),Line(
                points={{-90,0},{-40,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{40,0},{90,0}},
                color={0,0,0},
                thickness=1),Line(
                points={{-40,0},{38,-28}},
                color={0,0,0},
                thickness=1),Line(
                points={{-40,0},{40,0}},
                color={0,0,0},
```

```
                pattern=LinePattern.Dot,
                thickness=1)}),
      Diagram(graphics),
      DymolaStoredErrors);
  end IdealISwitch;
end Semiconductors;

package Devices

  package Converters

    model BackEMF
      "Counter-electromotive force"
      Interfaces.PosPin Pp annotation (Placement(
            transformation(extent={{-8,88},{12,108}}),
            iconTransformation(extent={{-8,88},{12,108}})));
      Interfaces.NegPin Np annotation (Placement(
            transformation(extent={{-10,-110},{10,-90}}),
            iconTransformation(extent={{-10,-110},{10,-90}})));
      Interfaces.MechanicalAxis mechanicalAxis
        annotation (Placement(transformation(extent={{-10,
              -110},{10,-90}}), iconTransformation(
            extent={{90,-10},{110,10}})));
      parameter Real Ke(unit="V.s/rad") = 0.0064
        "Back emf constant";
      parameter Real Kt(unit="N.m/A") = 0.0065
        "Torque constant";
      parameter SI.RotationalDampingConstant bm=4.121*1e-6
        "Friction constant";
      parameter Modelica.Units.SI.Inertia J=3.87*1e-7;
      Modelica.Units.SI.Voltage Vemf;
      Modelica.Units.SI.Current i=Pp.i;
      Modelica.Units.SI.Torque Te "Electrical torque";
      Modelica.Units.SI.Torque Tb "Friction torque";
      Modelica.Units.SI.Torque Tload=mechanicalAxis.T
        "Axis torque";
      Modelica.Units.SI.Angle Phi=mechanicalAxis.Phi;
      Modelica.Units.SI.AngularVelocity w=der(Phi);
      constant Real pi=Modelica.Constants.pi;

    equation
      Pp.i + Np.i = 0;
      Vemf = Ke*w;
      Vemf = Pp.v - Np.v;
      Te = i*Kt;
      Tb = bm*w;
      Te - Tb - Tload = J*der(w);
      annotation (Icon(coordinateSystem(
              preserveAspectRatio=false), graphics={
              Rectangle(
                      extent={{-30,80},{30,60}},
                      lineColor={28,108,200},
                      fillColor={215,215,215},
                      fillPattern=FillPattern.Solid),
              Rectangle(
                      extent={{-30,-60},{30,-80}},
                      lineColor={28,108,200},
                      fillColor={0,128,255},
                      fillPattern=FillPattern.Solid),
              Ellipse(extent={{-50,50},{50,-50}},
                      lineColor={28,108,200},
```

```
                fillColor={135,135,135},
                fillPattern=FillPattern.Solid),
          Line(   points={{60,-40},{60,40}},
                color={28,108,200},
                smooth=Smooth.Bezier,
                arrow={Arrow.None,Arrow.Open},
                thickness=1),Line(
                points={{-60,40},{-60,-40}},
                color={28,108,200},
                smooth=Smooth.Bezier,
                arrow={Arrow.None,Arrow.Open},
                thickness=1)}), Diagram(
        coordinateSystem(preserveAspectRatio=false)));


end BackEMF;

model ElectricConverter
  Interfaces.PosPin P_In annotation (Placement(
        transformation(extent={{-110,50},{-90,70}}),
        iconTransformation(extent={{-110,50},{-90,70}})));
  Interfaces.NegPin N_In annotation (Placement(
        transformation(extent={{-110,-70},{-90,-50}}),
        iconTransformation(extent={{-110,-70},{-90,-50}})));
  Interfaces.PosPin P_Out annotation (Placement(
        transformation(extent={{90,50},{110,70}}),
        iconTransformation(extent={{90,50},{110,70}})));
  Interfaces.NegPin N_Out annotation (Placement(
        transformation(extent={{90,-70},{110,-50}}),
        iconTransformation(extent={{90,-70},{110,-50}})));
  Interfaces.InPort DutyCycle annotation (Placement(
        transformation(
        extent={{-10,-10},{10,10}},
        rotation=270,
        origin={0,100}), iconTransformation(
        extent={{-10,-10},{10,10}},
        rotation=270,
        origin={0,100})));
  input SI.Voltage V_In=P_In.v - N_In.v;
  output SI.Voltage V_Out=P_Out.v - N_Out.v;
  input SI.Current I_In;
  // Input port current (Source side)
  output SI.Current I_Out=P_Out.i;
  // Output port curren (Load side)

  SI.Power PwIn=V_In*I_In;
  SI.Power PwOut=V_Out*I_Out;

  SI.Energy EBalance(start=0);
equation

  der(EBalance) = PwIn + PwOut;

  I_In = if DutyCycle >= 0 then P_In.i else -P_In.i;

  // Set V_Out
  V_Out = DutyCycle*V_In;

  PwIn + PwOut = 0;
```

```
    //Currents balnce
    P_In.i + N_In.i = 0;
    P_Out.i + N_Out.i = 0;

    annotation (Icon(coordinateSystem(
            preserveAspectRatio=false), graphics={
            Line(   points={{-100,-100},{100,100}},
                    color={0,0,0},
                    thickness=1),Text(
                    extent={{-80,80},{0,0}},
                    textColor={0,0,0},
                    textString="DC-IN"),Text(
                    extent={{0,0},{80,-80}},
                    textColor={0,0,0},
                    textString="DC-OUT"),Rectangle(
                    extent={{-100,100},{100,-100}},
                    lineColor={0,0,0},
                    lineThickness=1)}), Diagram(
          coordinateSystem(preserveAspectRatio=false)));

  end ElectricConverter;
end Converters;

package Machines

  model PMSM

    constant Real pi=Constant.pi;
    Interfaces.PosPin Vin[3] annotation (Placement(
          transformation(extent={{-110,-10},{-90,10}},
            rotation=0)));
    Interfaces.MechanicalAxis Rotor annotation (
        Placement(transformation(extent={{90,-10},{110,
              10}}, rotation=0)));
    parameter Integer Pp(min=1) = 2 "Poles pairs";
    parameter SI.Resistance Rs=2.98
      "Stator winding resistance";
    parameter SI.Inductance Ld=7e-3;
    parameter SI.Inductance Lq=7e-3;
    parameter SI.Inertia J=4.7e-5;
    parameter Real Bv(
      unit="N.m.s/rad",
      min=0) = 1.1e-4 " Dynamic viscosity";
    parameter SI.MagneticFlux Fmg=0.125;
    SI.Angle Th_e;
    SI.Angle Th_m;
    SI.AngularVelocity we
      "Electrical angular velocity";
    SI.AngularVelocity wm(start=0)
      "Mechanical angular velocity";
    SI.Torque Tl=Rotor.T;
    SI.Torque Te;
    SI.Current Ia=Vin[1].i;
    SI.Current Ib=Vin[2].i;
    SI.Current Ic=Vin[3].i;
    SI.Current Id "Current on direct axis";
    SI.Current Iq "Current on normal axis";
    SI.Voltage Va=smooth(1, Vin[1].v);
    SI.Voltage Vb=smooth(1, Vin[2].v);
    SI.Voltage Vc=smooth(1, Vin[3].v);
    SI.Voltage Vd;
```

```
  SI.Voltage Vq;
  SI.MagneticFlux Fd "Stator magnetic fluxes";
  SI.MagneticFlux Fq
    "Stator permanent magnetic fluxes";
  Real PT[2,3]=2/3*[cos(Th_e),cos(Th_e - 2*pi/3),
      cos(Th_e - 4*pi/3); -sin(Th_e),-sin(Th_e - 2*
      pi/3),-sin(Th_e - 4*pi/3)];
equation
  [Id; Iq] = PT*[Ia; Ib; Ic];
  [Vd; Vq] = PT*[Va; Vb; Vc];
  Ia + Ib + Ic = 0;

  Fd = Ld*Id + Fmg;
  Fq = Lq*Iq;

  Vq = Rs*Iq + we*Fd + der(Fq);
  Vd = Rs*Id - we*Fq + der(Fd);

  Te = 1.5*Pp*(Fd*Iq - Fq*Id);

  Te - Tl - Bv*wm = J*der(wm);


  we = Pp*wm;

  der(Th_e) = we;
  der(Th_m) = wm;

  Rotor.Phi = Th_m;
  annotation (Icon(graphics={Ellipse(
                  extent={{-80,80},{80,-80}},
                  lineColor={0,0,255},
                  fillColor={175,175,175},
                  fillPattern=FillPattern.Solid),
          Ellipse(extent={{-70,70},{70,-70}},
                  lineColor={0,0,255},
                  fillColor={255,255,255},
                  fillPattern=FillPattern.Solid),
          Rectangle(
                  extent={{-36,40},{-28,-40}},
                  lineColor={0,0,255},
                  fillColor={255,0,0},
                  fillPattern=FillPattern.Solid),
          Rectangle(
                  extent={{22,42},{30,-36}},
                  lineColor={0,0,255},
                  fillColor={0,0,255},
                  fillPattern=FillPattern.Solid)}),
      Diagram(graphics));


end PMSM;

model DCMotor

  constant Real pi=Constant.pi;
  Interfaces.PosPin Vp annotation (Placement(
        transformation(extent={{-108,50},{-88,70}},
          rotation=0), iconTransformation(extent={{-108,
            50},{-88,70}})));
```

```
  Interfaces.MechanicalAxis Rotor annotation (
      Placement(transformation(extent={{90,-10},{110,
            10}}, rotation=0)));


  Interfaces.NegPin Vn annotation (Placement(
        transformation(extent={{-110,-70},{-90,-50}},
          rotation=0), iconTransformation(extent={{-110,
          -70},{-90,-50}})));
  Basic.Resistance R1(R=Rm) annotation (Placement(
        transformation(
        extent={{-20,-20},{20,20}},
        rotation=180,
        origin={-60,60})));
  Basic.IdealCoil L1(L=Lm) annotation (Placement(
        transformation(
        extent={{-20,-20},{20,20}},
        rotation=180,
        origin={0,60})));


  parameter Modelica.Units.SI.Resistance Rm=0.837
    "Motor electrical resistance";
  parameter Modelica.Units.SI.Inductance Lm=0.0008
    "Motor electrical inductance";

  parameter Real Ke(unit="V.s/rad") = 0.0064
    "Back emf constant";
  parameter Real Kt(unit="N.m/A") = 0.0065
    "Torque constant";
  parameter
    Modelica.Units.SI.RotationalDampingConstant bm=4.121
      *1e-6 "Visc. friction constant";
  parameter Modelica.Units.SI.Inertia J=3.87*1e-7
    "Rotor's inertia";

  Converters.BackEMF backEMF(
    Ke=Ke,
    Kt=Kt,
    bm=bm,
    J=J) annotation (Placement(transformation(
          extent={{-20,-40},{60,40}})));
equation
  connect(backEMF.mechanicalAxis, Rotor)
    annotation (Line(
      points={{60,0},{100,0}},
      color={135,135,135},
      smooth=Smooth.Bezier,
      thickness=1));
  connect(R1.p, L1.n) annotation (Line(
      points={{-40,60},{-20,60}},
      color={0,0,255},
      thickness=0.5));
  connect(R1.n, Vp) annotation (Line(
      points={{-80,60},{-98,60}},
      color={0,0,255},
      thickness=0.5));
  connect(L1.p, backEMF.Pp) annotation (Line(
      points={{20,60},{20.8,60},{20.8,39.2}},
      color={0,0,255},
```

```
            thickness=0.5));
        connect(backEMF.Np, Vn) annotation (Line(
            points={{20,-40},{20,-60},{-100,-60}},
            color={0,0,255},
            thickness=0.5));
        annotation (Icon(graphics={Ellipse(
                        extent={{-80,80},{80,-80}},
                        lineColor={0,0,255},
                        fillColor={175,175,175},
                        fillPattern=FillPattern.Solid),
                    Ellipse(extent={{-70,70},{70,-70}},
                        lineColor={0,0,255},
                        fillColor={255,255,255},
                        fillPattern=FillPattern.Solid),
                    Rectangle(
                        extent={{-36,40},{-28,-40}},
                        lineColor={0,0,255},
                        fillColor={255,0,0},
                        fillPattern=FillPattern.Solid),
                    Rectangle(
                        extent={{22,42},{30,-36}},
                        lineColor={0,0,255},
                        fillColor={0,0,255},
                        fillPattern=FillPattern.Solid)}));

      end DCMotor;

    end Machines;

    model CurrentSaturation

      input Interfaces.NegPin In annotation (Placement(
            transformation(extent={{-110,-10},{-90,10}},
              rotation=0)));
      output Interfaces.PosPin Out annotation (Placement(
            transformation(extent={{90,-12},{110,8}},
              rotation=0)));
      parameter Real IMax=1e6;
      parameter Real IMin=-1e-6;

      SI.Current Iout=Out.i;
      SI.Current Iin=In.i;
    equation
      Iout = min(IMax, max(IMin, Iin));
      Out.v = In.v;
      annotation (Icon(graphics={Line(
                    points={{-80,80},{80,80}},
                    color={0,0,255}),Line(
                    points={{-80,-80},{80,-80},{80,-80}},
                    color={0,0,255}),Line(
                    points={{-80,-80},{-74,-80},{-60,-80},
                {-20,80},{14,80},{40,-76},{40,-80},{60,-80},
                {76,-6}},
                    color={0,0,0},
                    thickness=0.5)}), Diagram(graphics));

    end CurrentSaturation;
  end Devices;

end Electrical;
```

A5. Mechanical

```
package Mechanical
    package Basic
      model RotLoad

        Interfaces.MechanicalAxis Axis annotation (
            Placement(transformation(extent={{-10,88},{10,108}},
                rotation=0)));
        parameter SI.Inertia J=10 "Inertial load";
        parameter SI.DynamicViscosity fs=8.5e-6
          " Dynamic viscosity";
        constant Real eps=1e-3;
        SI.AngularVelocity w;
      equation
        der(Axis.Phi) = w;
        J*der(w) = -Axis.T - w*fs;
        annotation (Icon(graphics={Line(
                      points={{0,88},{0,2},{0,0}},
                      color={0,0,255}),Ellipse(
                      extent={{-80,-20},{80,-2}},
                      lineColor={0,0,255},
                      fillColor={175,175,175},
                      fillPattern=FillPattern.Solid),Line(
                      points={{-56,-28},{42,-28},{50,-28}},
                      color={0,0,0},
                      thickness=0.5,
                      arrow={Arrow.None,Arrow.Filled})}));

      end RotLoad;

      model RotAxis

        Interfaces.MechanicalAxis AxisA annotation (
            Placement(transformation(extent={{-110,-10},{-90,
                  10}}, rotation=0)));
        Interfaces.MechanicalAxis AxisB annotation (
            Placement(transformation(extent={{88,-10},{108,10}},
                rotation=0)));
        parameter SI.Inertia J=10 "Inertial load";
        SI.AngularVelocity w;
        SI.Angle Phi;
      equation
        der(Phi) = w;
        J*der(w) = AxisA.T + AxisB.T;
        AxisA.Phi = Phi;
        AxisB.Phi = Phi;
        annotation (Icon(graphics={Rectangle(
                      extent={{-92,-6},{88,6}},
                      lineColor={95,95,95},
                      lineThickness=0.5,
                      fillColor={255,255,255},
                      fillPattern=FillPattern.Backward)}),
            Diagram(graphics));
```

```
end RotAxis;

model Wheel
  constant Real pi=Constant.pi;
  Interfaces.MechanicalAxis Axis annotation (
      Placement(transformation(extent={{-10,88},{10,108}},
          rotation=0)));
  parameter SI.Inertia J=10 "Inertial load";
  parameter SI.Length R=0.3 "Radius";
  parameter Real fs(
    unit="N.m.s/rad",
    min=0) = 8.5e-6 " Dynamic viscosity";
  parameter SI.Mass M=1200 "Vehicle mass";
  constant Real eps=1e-3;
  SI.AngularVelocity w;
  SI.Velocity V(start=0);
  Interfaces.OutPort Vwheel annotation (Placement(
        transformation(
        origin={-60,100},
        extent={{-10,-10},{10,10}},
        rotation=90)));
equation
  der(Axis.Phi) = w;

  V = 2*pi*R*w;

  if abs(V) < eps and abs(Axis.T) < eps then
    //Stacionary state
    der(w) = -1e3*w;
  else
    Axis.T + w*fs + (J + M*R^2)*der(w) = 0;

  end if;
  Vwheel = V;
  annotation (Icon(graphics={Line(
                points={{0,88},{0,2},{0,0}},
                color={0,0,255}),Ellipse(
                extent={{-70,-70},{70,70}},
                lineColor={0,0,255},
                fillColor={0,0,0},
                fillPattern=FillPattern.Solid),Line(
                points={{-50,-94},{48,-94},{56,-94}},
                color={0,0,0},
                thickness=0.5,
                arrow={Arrow.None,Arrow.Filled}),
          Ellipse(
                extent={{-60,60},{60,-60}},
                lineColor={215,215,215},
                pattern=LinePattern.None,
                fillColor={255,255,255},
                fillPattern=FillPattern.CrossDiag),
          Ellipse(
                extent={{-10,10},{10,-10}},
                lineColor={215,215,215},
                fillColor={175,175,175},
                fillPattern=FillPattern.CrossDiag)}));

end Wheel;

model FixedTorque
```

```
  Interfaces.MechanicalAxis Axis annotation (
      Placement(transformation(extent={{-10,88},{10,108}},
          rotation=0)));
  parameter SI.Torque T=10 "Torque";

equation

  T = Axis.T annotation (Icon(graphics={Line(
                points={{0,88},{0,2},{0,0}},
                color={0,0,255}),Ellipse(
                extent={{-80,-20},{80,-2}},
                lineColor={0,0,255},
                fillColor={175,175,175},
                fillPattern=FillPattern.Backward,
                startAngle=0,
                endAngle=360),Line(
                points={{-56,-28},{42,-28},{50,-28}},
                color={0,0,0},
                thickness=0.5,
                arrow={Arrow.None,Arrow.Filled}),
          Rectangle(
                extent={{-4,88},{4,-2}},
                lineColor={0,0,255},
                lineThickness=0.5,
                fillColor={135,135,135},
                fillPattern=FillPattern.Solid)}));

  annotation (Icon(graphics={Rectangle(
                extent={{-4,88},{4,0}},
                lineColor={0,0,255},
                lineThickness=0.5,
                fillColor={135,135,135},
                fillPattern=FillPattern.Solid),
          Rectangle(
                extent={{-20,0},{22,-40}},
                lineColor={0,0,255},
                lineThickness=0.5,
                fillColor={135,135,135},
                fillPattern=FillPattern.CrossDiag)}));
end FixedTorque;

model FixedSpeed

  Interfaces.MechanicalAxis Axis annotation (
      Placement(transformation(extent={{-10,88},{10,108}},
          rotation=0)));
  parameter SI.AngularVelocity w=1 "Angular velocity";
equation
  der(Axis.Phi) = w;

  annotation (Icon(graphics={Line(
                points={{0,88},{0,2},{0,0}},
                color={0,0,255}),Ellipse(
                extent={{-80,-20},{80,-2}},
                lineColor={0,0,255},
                fillColor={175,175,175},
                fillPattern=FillPattern.CrossDiag),
          Line( points={{-56,-28},{42,-28},{50,-28}},
                color={0,0,0},
                thickness=0.5,
```

```
                arrow={Arrow.None,Arrow.Filled})}));

  end FixedSpeed;

  model Gearbox

    input Interfaces.MechanicalAxis AxisA annotation (
        Placement(transformation(extent={{-112,-10},{-92,
              10}}, rotation=0), iconTransformation(
            extent={{-112,-10},{-92,10}})));
    output Interfaces.MechanicalAxis AxisB annotation (
        Placement(transformation(extent={{88,-10},{108,10}},
            rotation=0)));
    parameter Real N(min=0.01) = 1 "Conversion ratio";

    SI.Torque TIn=AxisA.T;
    SI.Torque TOut=AxisB.T;


    SI.Angle PhiIn=AxisA.Phi;
    SI.Angle PhiOut=AxisB.Phi;
  equation

    der(PhiOut) = der(N*PhiIn);
    TIn + N*TOut = 0;

    annotation (Icon(graphics={Rectangle(
                  extent={{-92,-6},{88,6}},
                  lineColor={95,95,95},
                  lineThickness=0.5,
                  fillColor={255,255,255},
                  fillPattern=FillPattern.Backward)}),
        Diagram(graphics));

  end Gearbox;
end Basic;

model BodyFrame1DOF
  "1 degree of freedom body frame"

  Interfaces.MechanicalAxis TorqueIN annotation (
      Placement(transformation(extent={{-110,50},{-90,70}},
          rotation=0)));
  output Interfaces.OutPort V "Vehicle speed"
    annotation (Placement(transformation(extent={{90,50},
            {110,70}}, rotation=0)));
  input Interfaces.InPort Alpha "Terrain slope"
    annotation (Placement(transformation(extent={{-110,-50},
            {-90,-30}}, rotation=0)));
  constant SI.Acceleration g=Modelica.Constants.g_n;
  constant Real pi=Modelica.Constants.pi;
  parameter SI.Length R(min=0.01) = 0.25 "Wheel radius";
  parameter SI.Mass M=1500 "Vehicle mass";
  parameter SI.Area Af=2 "Vehicle front area";
  parameter Real Cd(min=0) = 0.248 "Vehicle drag coef.";
  parameter Real Cr(min=0) = 0.01
    "Tyres rolling resistance coef.";
  parameter SI.Density rho(displayUnit="kg/m3") = 1.2
    "Air density";

  SI.Force F "Powertrain provided force";
```

```
  SI.Force Fd "Drag force";
  SI.Force Fr "Rolling resistance";
  SI.Force Fg "Weight poryected force";
  SI.Force Fi "Inertial force";

equation
  Fd = 0.5*Cd*Af*rho*(V^2)*sign(V);
  Fr = M*g*cos(Alpha*pi/180)*Cr;
  Fg = M*g*sin(Alpha*pi/180);
  Fi = M*der(V);

  F - Fd - Fr - Fg - Fi = 0;
  F = -TorqueIN.T/R;
  der(TorqueIN.Phi) = V/(2*pi*R);

  annotation (Icon(graphics={Rectangle(
              extent={{-100,100},{100,-100}},
              lineColor={0,0,255},
              fillColor={241,241,241},
              fillPattern=FillPattern.Solid),Text(
              extent={{-60,100},{60,-20}},
              textColor={0,0,255},
              textString="BODY"),Text(
              extent={{-60,40},{60,-80}},
              textColor={0,0,255},
              textString="1 DOF")}), Diagram(graphics));


end BodyFrame1DOF;

model Slope

  Interfaces.InPort H "Current Heigth" annotation (
      Placement(transformation(extent={{-110,50},{-90,70}},
          rotation=0)));
  Interfaces.OutPort Alpha "Slope" annotation (
      Placement(transformation(extent={{90,-10},{110,10}},
          rotation=0)));

  Interfaces.InPort d "Current displacement"
    annotation (Placement(transformation(extent={{-110,-50},
            {-90,-30}}, rotation=0)));


equation
  when (time > 0) then
    Alpha = asin(der(H)/der(d));
  end when annotation (Diagram(graphics={Text(
              extent={{-60,-60},{-80,-40}},
              textColor={28,108,200},
              textString="d"),Text(
              extent={{-60,40},{-80,60}},
              textColor={28,108,200},
              textString="H")}), Icon(graphics={
          Rectangle(
              extent={{-100,100},{100,-100}},
              lineColor={28,108,200}),Line(
              points={{-60,-60},{60,-60}},
              color={28,108,200},
              thickness=1,
              arrow={Arrow.None,Arrow.Filled}),Line(
```

```
                points={{-60,-60},{60,60}},
                color={28,108,200},
                thickness=1,
                arrow={Arrow.None,Arrow.Filled}),Line(
                points={{60,60},{60,-60}},
                color={28,108,200},
                thickness=1,
                arrow={Arrow.Filled,Arrow.None}),Text(
                extent={{-14,-32},{-46,-62}},
                textColor={28,108,200},
                textString="α"),Text(
                extent={{-80,92},{-100,72}},
                textColor={28,108,200},
                textString="y"),Text(
                extent={{-80,-8},{-100,-28}},
                textColor={28,108,200},
                textString="x"),Text(
                extent={{20,-62},{0,-82}},
                textColor={28,108,200},
                textString="Δx"),Text(
                extent={{82,0},{62,-20}},
                textColor={28,108,200},
                textString="Δy")}));
    end Slope;
  end Mechanical;
```

A6. Control

```
package Control
    model PID

      Interfaces.OutPort Out annotation (Placement(
            transformation(extent={{90,8},{110,28}},
              rotation=0)));
      Interfaces.InPort In annotation (Placement(
            transformation(extent={{-110,-30},{-90,-10}},
              rotation=0)));
      Interfaces.InPort Ref annotation (Placement(
            transformation(extent={{-110,30},{-90,50}},
              rotation=0)));


      parameter Boolean LimitOut=false
        annotation (choices(checkBox=true));
      parameter Real Max=0;
      parameter Real Min=0;

      parameter Boolean DeadZone=false
        annotation (choices(checkBox=true));
      parameter Real eps(min=0) = 1e-9 "Dead zone range";

      parameter Real K=1;
      parameter Real I=0;
      parameter Real D=0;

      parameter Real Smt=1e6 "Smooth factor";

      Real AuxOut(start=0);
      Real Error(start=0);
```

```
  Real IntEr(start=0);

equation
  AuxOut = K*Error + D*der(Error) + I*IntEr;
  Error = Ref - In;

  // DEADZONE
  if DeadZone == true and noEvent(abs(AuxOut) < eps)
       then
    der(Out) = -Smt*Out;
    der(IntEr) = 0;
  elseif LimitOut == true then
    //ANTIWINDUP
    if noEvent(AuxOut >= Max) then
      der(Out) = Smt*(Max - Out);
      der(IntEr) = 0;
    elseif noEvent(AuxOut <= Min) then
      der(Out) = Smt*(Min - Out);
      der(IntEr) = 0;
    else
      der(Out) = Smt*(AuxOut - Out);
      der(IntEr) = Error;
    end if;
  else
    der(Out) = Smt*(AuxOut - Out);
    der(IntEr) = Error;
  end if;

  annotation (Icon(graphics={Text(
          extent={{80,-40},{-80,60}},
          textColor={0,0,255},
          textString="PID"), Rectangle(extent={{-100,80},
              {100,-60}}, lineColor={0,0,255})}),
      DymolaStoredErrors);


end PID;

model PI

  Interfaces.OutPort Out annotation (Placement(
        transformation(extent={{90,8},{110,28}},
          rotation=0)));
  Interfaces.InPort In annotation (Placement(
        transformation(extent={{-110,-30},{-90,-10}},
          rotation=0)));
  Interfaces.InPort Ref annotation (Placement(
        transformation(extent={{-110,30},{-90,50}},
          rotation=0)));
  parameter Real K=1;
  parameter Real I=0;
  Real Error;
  Real IntEr(start=0);



initial equation
  IntEr = 0;
```

```
    equation
      Out = K*Error + I*IntEr;
      Error = Ref - In;
      der(IntEr) = Error;

      annotation (Icon(graphics={Text(
              extent={{80,-40},{-80,60}},
              textColor={0,0,255},
              textString="PI"), Rectangle(extent={{-100,80},
                  {100,-60}}, lineColor={0,0,255})}),
          DymolaStoredErrors);

    end PI;

  end Control;
```

A7. Sensors

```
package Sensors
    model Vsensor

      EPowertrain.Interfaces.PosPin p annotation (Placement(
            transformation(extent={{-110,-10},{-90,10}},
              rotation=0)));
      EPowertrain.Interfaces.PosPin n annotation (Placement(
            transformation(extent={{90,-10},{110,10}},
              rotation=0)));

      Modelica.Units.SI.Voltage v;
      Interfaces.OutPort outPort annotation (Placement(
            transformation(
            origin={0,100},
            extent={{-10,-10},{10,10}},
            rotation=90)));
    equation
      p.i = 0;
      n.i = 0;
      v = p.v - n.v;
      outPort = v;
      annotation (Diagram(graphics), Icon(graphics={
            Rectangle(
              extent={{-74,6},{-34,-6}},
              lineColor={0,0,0},
              lineThickness=1,
              fillColor={0,0,0},
              fillPattern=FillPattern.Solid),
            Rectangle(
              extent={{-60,20},{-48,-20}},
              lineColor={0,0,0},
              lineThickness=1,
              fillColor={0,0,0},
              fillPattern=FillPattern.Solid),
            Rectangle(
              extent={{68,20},{54,-20}},
              lineColor={0,0,0},
              lineThickness=1,
              fillColor={0,0,0},
              fillPattern=FillPattern.Solid),
            Ellipse(extent={{-80,80},{80,-80}}, lineColor={0,
```

```
              0,0})}));
end Vsensor;

model AxialSpeed

  Interfaces.MechanicalAxis Axis_In annotation (
      Placement(transformation(extent={{-10,90},{10,110}},
          rotation=0)));
  Interfaces.MechanicalAxis Axis_Out annotation (
      Placement(transformation(extent={{-10,-110},{10,-90}},
          rotation=0)));
  Interfaces.OutPort Wm annotation (Placement(
        transformation(extent={{90,70},{110,90}},
          rotation=0)));
  Interfaces.OutPort Th_m annotation (Placement(
        transformation(extent={{90,-50},{110,-30}},
          rotation=0)));
  Interfaces.OutPort We annotation (Placement(
        transformation(extent={{90,32},{110,52}},
          rotation=0)));
  Interfaces.OutPort Th_e annotation (Placement(
        transformation(extent={{90,-90},{110,-70}},
          rotation=0)));
  parameter Integer N(min=1) "Pole pairs";
  constant Real pi=Constant.pi;



equation

  Th_e = N*Axis_In.Phi;
  Th_m = Axis_In.Phi;

  when (Th_e >= 2*pi) then
    reinit(Th_e, 0);
  end when;


  Wm = der(Th_e);
  We = der(Th_m);
  connect(Axis_In, Axis_Out) annotation (Line(points={{0,
          100},{0,-100}}, color={0,0,255}));
  annotation (
    Icon(graphics={
        Rectangle(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,255},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid),
        Text(
          extent={{20,110},{80,50}},
          textColor={0,0,255},
          textString="Wm"),
        Text(
          extent={{20,-10},{80,-70}},
          textColor={0,0,255},
          textString="Th_m"),
        Text(
          extent={{22,72},{82,12}},
          textColor={0,0,255},
          textString="We"),
```

```
        Text(
          extent={{22,-48},{82,-108}},
          textColor={0,0,255},
          textString="Th_e"),
        Rectangle(extent={{-92,100},{-98,100}},
            lineColor={28,108,200})}),
    Diagram(graphics),
    DymolaStoredErrors);

end AxialSpeed;

model CurrentSensor

  Interfaces.PosPin P annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}},
          rotation=0)));
  Interfaces.NegPin N annotation (Placement(
        transformation(extent={{90,-12},{110,8}},
          rotation=0)));
  Interfaces.OutPort Imeas annotation (Placement(
        transformation(extent={{-10,88},{10,108}},
          rotation=0)));


equation
  P.v = N.v;
  N.i = Imeas;
  P.i + N.i = 0;
  annotation (
    Icon(graphics={
        Ellipse(
          extent={{-60,60},{60,-62}},
          lineColor={0,0,0},
          lineThickness=0.5),
        Text(
          extent={{-40,40},{40,-40}},
          textColor={0,0,0},
          textString="A"),
        Line(
          points={{-90,0},{-60,0}},
          color={0,0,0},
          thickness=0.5),
        Line(
          points={{60,0},{90,0}},
          color={0,0,0},
          thickness=0.5),
        Line(
          points={{0,86},{0,60}},
          color={0,0,0},
          pattern=LinePattern.Dash,
          thickness=0.5)}),
    DymolaStoredErrors,
    Diagram(graphics));


end CurrentSensor;

model Encoder

  Interfaces.MechanicalAxis Axis_In annotation (
      Placement(transformation(extent={{-10,90},{10,110}},
```

```
            rotation=0)));
    Interfaces.MechanicalAxis Axis_Out annotation (
        Placement(transformation(extent={{-10,-110},{10,-90}},
            rotation=0)));
    Interfaces.OutPort Th_m annotation (Placement(
          transformation(extent={{90,40},{110,60}},
            rotation=0)));
    Interfaces.OutPort Th_e annotation (Placement(
          transformation(extent={{90,-60},{110,-40}},
            rotation=0)));
    parameter Integer N(min=1) "Pole pairs";
    constant Real pi=Constant.pi;

    Real Th_e_raw;
    Real Th_m_raw;


    Real Th_e_event(start=0);
    Real Th_m_event(start=0);

  equation

    Th_m_raw = Axis_In.Phi;
    Th_e_raw = N*Axis_In.Phi;


    Th_m_event = Th_m_raw;
    Th_e_event = Th_e_raw;


    Th_e = smooth(1, Th_e_event);
    Th_m = smooth(1, Th_m_event);

    connect(Axis_In, Axis_Out) annotation (Line(points={{0,
            100},{0,-100}}, color={0,0,255}));
    annotation (
      Icon(graphics={
          Rectangle(extent={{-100,100},{100,-100}},
              lineColor={0,0,255}),
          Text(
            extent={{-80,100},{20,0}},
            textColor={0,0,255},
            textString="Th_m"),
          Text(
            extent={{-80,0},{20,-100}},
            textColor={0,0,255},
            textString="Th_e")}),
      Diagram(graphics={Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={28,108,200})})),
      DymolaStoredErrors);

  end Encoder;
end Sensors;
```

A8. Examples

```
package Examples
```

```
model UDDS_Cycle
  "Example experiment under UDDS drive cycle"
  Electrical.Devices.Converters.ElectricConverter
    electricConverter annotation (Placement(
        transformation(extent={{194,-24},{214,-4}})));
  Mechanical.BodyFrame1DOF bodyFrame1DOF(
    M=1500,
    Af=2.2,
    Cd=0.29,
    Cr=0.009,
    rho=1.2) annotation (Placement(transformation(
          extent={{326,-84},{346,-64}})));
  Electrical.Sources.Ground ground2 annotation (
      Placement(transformation(extent={{234,-74},{254,-54}})));
  Electrical.Sources.Ground ground1 annotation (
      Placement(transformation(extent={{128,-60},{148,-40}})));
  Control.PID pID(
    LimitOut=false,
    Max=1,
    Min=-1,
    K=20,
    I=0.1) annotation (Placement(transformation(extent={
          {132,4},{152,24}})));
  Sources.UDDS uDDS annotation (Placement(
        transformation(extent={{2,22},{22,42}})));
  Modelica.Blocks.Math.Gain mph_to_ms(k=0.44704)
    annotation (Placement(transformation(extent={{62,22},
          {82,42}})));
  Sources.Constant Constant(Value=0) annotation (
      Placement(transformation(extent={{268,-90},{288,-70}})));
  Electrical.Sources.Battery battery(
    Cap=30,
    Vd=300,
    Vf=420,
    InitSOC=0.8) annotation (Placement(transformation(
          extent={{108,-34},{128,-14}})));
  Electrical.Devices.Machines.DCMotor dCMotor_2_1(
    Rm=0.025,
    Lm=1e-3,
    Ke=1.4,
    Kt=1.4) annotation (Placement(transformation(
        extent={{-10,-10},{10,10}},
        rotation=0,
        origin={306,-16})));
equation
  connect(ground2.p, electricConverter.N_Out)
    annotation (Line(points={{244,-54},{244,-36},{222,-36},
          {222,-20},{214,-20}}, color={0,0,255}));
  connect(pID.Out, electricConverter.DutyCycle)
    annotation (Line(points={{152,15.8},{204,15.8},{204,
          -4}}, color={0,0,0}));
  connect(uDDS.y, mph_to_ms.u) annotation (Line(points={
          {23,32},{60,32}}, color={0,0,127}));
  connect(mph_to_ms.y, pID.Ref) annotation (Line(points
        ={{83,32},{114,32},{114,18},{132,18}}, color={0,
          0,127}));
  connect(bodyFrame1DOF.V, pID.In) annotation (Line(
        points={{346,-68},{360,-68},{360,-66},{374,-66},
          {374,-98},{48,-98},{48,12},{132,12}}, color={0,
          0,0}));
```

```
  connect(Constant.Out, bodyFrame1DOF.Alpha)
    annotation (Line(points={{288,-79.8},{320,-79.8},{320,
          -78},{326,-78}}, color={0,0,0}));
  connect(battery.posPin, electricConverter.P_In)
    annotation (Line(points={{118,-14},{118,-8},{194,-8}},
        color={0,0,255}));
  connect(battery.negPin, ground1.p) annotation (Line(
        points={{118,-34},{118,-68},{154,-68},{154,-34},
          {138,-34},{138,-40}}, color={0,0,255}));
  connect(battery.negPin, electricConverter.N_In)
    annotation (Line(points={{118,-34},{118,-68},{154,-68},
          {154,-20},{194,-20}}, color={0,0,255}));
  connect(electricConverter.P_Out, dCMotor_2_1.Vp)
    annotation (Line(points={{214,-8},{290,-8},{290,-10},
          {296.2,-10}}, color={0,0,255}));
  connect(electricConverter.N_Out, dCMotor_2_1.Vn)
    annotation (Line(points={{214,-20},{222,-20},{222,-36},
          {290,-36},{290,-22},{296,-22}}, color={0,0,255}));
  connect(dCMotor_2_1.Rotor, bodyFrame1DOF.TorqueIN)
    annotation (Line(points={{316,-16},{316,-68},{326,-68}},
        color={0,0,255}));
  annotation (
    experiment(
      StopTime=1400,
      __Dymola_NumberOfIntervals=50000,
      Tolerance=0.01,
      __Dymola_Algorithm="Dassl"),
    Diagram(coordinateSystem(extent={{-100,-200},{580,100}})),
    Icon(coordinateSystem(extent={{-100,-200},{580,100}})));
end UDDS_Cycle;

model Trip "Example experiment"
  Electrical.Devices.Converters.ElectricConverter
    electricConverter annotation (Placement(
        transformation(extent={{196,-24},{216,-4}})));
  Mechanical.BodyFrame1DOF bodyFrame1DOF(
    R=0.29,
    M=1280,
    Af=2.38,
    Cd=0.29,
    Cr=0.0084,
    rho=1.225) annotation (Placement(transformation(
          extent={{324,-30},{344,-10}})));
  Electrical.Sources.Ground ground2 annotation (
      Placement(transformation(extent={{222,-60},{242,-40}})));
  Electrical.Sources.Ground ground1 annotation (
      Placement(transformation(extent={{114,-66},{134,-46}})));
  Control.PID Driver(
    LimitOut=true,
    Max=1,
    Min=-0.5,
    K=0.5,
    I=0.005,
    D=0.1,
    Out(start=1)) annotation (Placement(transformation(
          extent={{132,4},{152,24}})));
  Modelica.Blocks.Sources.CombiTimeTable Cycle(table=
        fill(0.0, 0, 2)) annotation (Placement(
        transformation(extent={{-60,8},{-40,28}})));
  Electrical.Sources.Battery battery(
    Cap=60,
```

```
    Vd=335,
    Vf=360,
    InitSOC=0.869,
    Rs=0.06,
    Imax=400) annotation (Placement(transformation(
          extent={{114,-32},{134,-12}})));
  Modelica.Blocks.Math.Gain kph2ms(k=1/3.6) annotation
    (Placement(transformation(extent={{20,8},{40,28}})));
  SignalRouting.Terminator DataSOC annotation (
      Placement(transformation(extent={{20,-20},{40,0}})));
  Mechanical.Slope slope annotation (Placement(
        transformation(extent={{254,-106},{274,-86}})));
  Modelica.Blocks.Continuous.Integrator integrator
    annotation (Placement(transformation(
        extent={{-10,-10},{10,10}},
        rotation=180,
        origin={338,-124})));
  SignalRouting.Terminator DataIbatt annotation (
      Placement(transformation(extent={{20,-40},{40,-20}})));
  SignalRouting.Terminator DataTorque annotation (
      Placement(transformation(extent={{20,-60},{40,-40}})));
  Mechanical.Basic.Gearbox gearbox1(N=1/6) annotation (
      Placement(transformation(extent={{288,-24},{308,-4}})));
  Electrical.Basic.Resistance RLoad(R=80.4765)
    annotation (Placement(transformation(
        extent={{-9,-10},{9,10}},
        rotation=90,
        origin={185,-44})));
  Electrical.Devices.Machines.DCMotor dCMotor(
    Rm=1.72,
    Lm=106.26e-6,
    Ke=0.7144,
    Kt=0.72,
    bm=5e-4,
    J=31e-3) annotation (Placement(transformation(
        extent={{-10,-10},{10,10}},
        rotation=0,
        origin={260,-14})));
equation
  connect(dCMotor.Vp, electricConverter.P_Out)
    annotation (Line(points={{250.2,-8},{216,-8}},
        color={0,0,255}));
  connect(ground2.p, electricConverter.N_Out)
    annotation (Line(points={{232,-40},{232,-20},{216,-20}},
        color={0,0,255}));
  connect(Driver.Out, electricConverter.DutyCycle)
    annotation (Line(points={{152,15.8},{206,15.8},{206,
        -4}}, color={0,0,0}));
  connect(dCMotor.Vn, electricConverter.N_Out)
    annotation (Line(points={{250,-20},{216,-20}},
        color={0,0,255}));
  connect(bodyFrame1DOF.V, Driver.In) annotation (Line(
        points={{344,-14},{358,-14},{358,-68},{102,-68},
          {102,12},{132,12}}, color={0,0,0}));
  connect(battery.posPin, electricConverter.P_In)
    annotation (Line(points={{124,-12},{124,-8},{196,-8}},
        color={0,0,255}));
  connect(battery.negPin, ground1.p) annotation (Line(
        points={{124,-32},{124,-46}}, color={0,0,255}));
  connect(battery.negPin, electricConverter.N_In)
    annotation (Line(points={{124,-32},{124,-36},{154,-36},
```

```
          {154,-20},{196,-20}}, color={0,0,255}));
    connect(kph2ms.y, Driver.Ref) annotation (Line(points
        ={{41,18},{132,18}}, color={0,0,127}));
    connect(Cycle.y[1], kph2ms.u) annotation (Line(points
        ={{-39,18},{18,18}}, color={0,0,127}));
    connect(DataSOC.inPort, Cycle.y[7]) annotation (Line(
        points={{19.8,-10},{-32,-10},{-32,18},{-39,18}},
        color={0,0,0}));
    connect(slope.Alpha, bodyFrame1DOF.Alpha) annotation
      (Line(points={{274,-96},{310,-96},{310,-24},{324,-24}},
        color={0,0,0}));
    connect(Cycle.y[2], slope.H) annotation (Line(points={
          {-39,18},{-32,18},{-32,-90},{254,-90}}, color
        ={0,0,127}));
    connect(integrator.y, slope.d) annotation (Line(
        points={{327,-124},{248,-124},{248,-100},{254,-100}},
        color={0,0,127}));
    connect(integrator.u, bodyFrame1DOF.V) annotation (
      Line(points={{350,-124},{358,-124},{358,-14},{344,
          -14}}, color={0,0,127}));
    connect(DataIbatt.inPort, Cycle.y[6]) annotation (
      Line(points={{19.8,-30},{-32,-30},{-32,18},{-39,18}},
        color={0,0,0}));
    connect(DataTorque.inPort, Cycle.y[4]) annotation (
      Line(points={{19.8,-50},{-32,-50},{-32,18},{-39,18}},
        color={0,0,0}));
    connect(dCMotor.Rotor, gearbox1.AxisA) annotation (
      Line(points={{270,-14},{287.8,-14}}, color={0,0,255}));
    connect(gearbox1.AxisB, bodyFrame1DOF.TorqueIN)
      annotation (Line(points={{307.8,-14},{324,-14}},
        color={0,0,255}));
    connect(RLoad.n, electricConverter.P_In) annotation (
      Line(points={{185,-35},{185,-8},{196,-8}}, color={
          0,0,255}));
    connect(RLoad.p, electricConverter.N_In) annotation (
      Line(points={{185,-53},{154,-53},{154,-36},{152,-36},
          {152,-20},{196,-20}}, color={0,0,255}));
    annotation (
      experiment(
        StopTime=1000,
        Tolerance=1e-05,
        __Dymola_Algorithm="Dassl"),
      Diagram(coordinateSystem(extent={{-100,-200},{580,100}}),
        graphics={Text(
              extent={{-94,30},{-60,8}},
              textColor={28,108,200},
              horizontalAlignment=TextAlignment.Left,
              textString="output:
1 Vel[kph]
2 Elevation[m]
3 Throtle [-]
4 Torque [Nm]
5 V batt [V]
6 I batt [A]
7 SoC [%]"),Text( extent={{364,-26},{370,-34}},
              textColor={28,108,200},
              textString="V"),Text(
              extent={{262,-128},{304,-136}},
              textColor={28,108,200},
              textString="Displacement"),Text(
              extent={{276,-80},{308,-90}},
```

```
                textColor={28,108,200},
                textString="Slope"),Text(
                extent={{70,30},{82,20}},
                textColor={28,108,200},
                textString="Vref"),Text(
                extent={{42,-8},{54,-18}},
                textColor={28,108,200},
                textString="SoC"),Text(
                extent={{42,-26},{54,-36}},
                textColor={28,108,200},
                textString="Ibatt"),Text(
                extent={{40,-46},{60,-58}},
                textColor={28,108,200},
                textString="Torque"),Text(
                extent={{176,-78},{218,-86}},
                textColor={28,108,200},
                textString="Height")}),
    Icon(coordinateSystem(extent={{-100,-200},{580,100}})));
end Trip;

model DCMotor
  Electrical.Devices.Machines.DCMotor Moments(
    Rm=30.9,
    Lm=0.803,
    Ke=1.323,
    Kt=1.323,
    bm=0.0005,
    J=0.0031) annotation (Placement(transformation(
          extent={{30,2},{50,22}})));
  Electrical.Sources.VStep vStep(
    St=0.5,
    v0=60,
    vf=248) annotation (Placement(transformation(
        extent={{-9,-10},{9,10}},
        rotation=270,
        origin={-75,30})));
  Electrical.Sources.Ground ground annotation (
      Placement(transformation(extent={{-64,-24},{-44,-4}})));
  Electrical.Devices.Machines.DCMotor Pasek(
    Rm=30.9,
    Lm=0.438,
    Ke=1.323,
    Kt=1.323,
    bm=0.0005,
    J=0.0036) annotation (Placement(transformation(
          extent={{-22,66},{-2,86}})));
equation
  connect(vStep.n, Moments.Vn) annotation (Line(points={
          {-75,21},{-76,21},{-76,4},{-72,4},{-72,6},{30,
          6}}, color={0,0,255}));
  connect(vStep.p, Moments.Vp) annotation (Line(points={
          {-75,39},{-76,39},{-76,48},{6,48},{6,18},{30.2,
          18}}, color={0,0,255}));
  connect(ground.p, Moments.Vn) annotation (Line(points
        ={{-54,-4},{-54,6},{30,6}}, color={0,0,255}));
  connect(Pasek.Vp, vStep.p) annotation (Line(points={{-21.8,
          82},{-75,82},{-75,39}}, color={0,0,255}));
  connect(Pasek.Vn, vStep.n) annotation (Line(points={{-22,
          70},{-26,70},{-26,6},{-72,6},{-72,4},{-76,4},{
          -76,21},{-75,21}}, color={0,0,255}));
  annotation (
```

```
      Icon(coordinateSystem(preserveAspectRatio=false)),
      Diagram(coordinateSystem(preserveAspectRatio=false)),
      experiment(__Dymola_Algorithm="Dassl"));
  end DCMotor;

end Examples
```

Appendix B – EPowertrain Library Documentation

EPowetrain

This is the main package of the EPowertrain library. It contains all components, models, and interfaces necessary to simulate the energy behavior of electric vehicle powertrains under configurable conditions.

Package Content

Name	Description
Interfaces	Contains standard interface connectors used across the library for electric, mechanical, and control domains. Ensures physical consistency and plug-and-play compatibility between components.
SignalRouting	Provides connectors, multiplexers, and demultiplexers for managing signal flow in large models or control systems.
Sources	Contains signal and power sources, including constant, step, sine and user-defined input profiles.
Electrical	Includes electrical components such as resistors, capacitors, inductors, diodes, and configurable sources. These are used as building blocks for more complex models.
Mechanical	Provides mechanical elements for translational and rotational motion, including inertia, load models, and coupling flanges for drivetrain representation.
Control	Contains control blocks used to implement energy management or actuator control strategies.
Sensors	Contains idealized sensors for measuring physical variables (e.g., voltage, current, torque, speed), used in control feedback or performance monitoring.
Examples	Provides usage examples and test benches for validating component behavior under specific driving conditions or scenarios.

EPowetrain.Interfaces

Contains standard interface connectors used across the library for electric, mechanical, and control domains. Ensures physical consistency and plug-and-play compatibility between components.

Package Content

Name	Description
PosPin	Electrical Pin (Positive)
NegPin	Electrical Pin (Negative)
ElectricPort	An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.
MechanicalAxis	Mechanical axis coupling
IO_Port	Input/output generalist port
BoolOutPort	Boolean output
BoolInPort	Boolean input
OutPort	Generalist output
InPort	Generalist input
ThreePins	Triple electrical port used as base for transistors

EPowetrain.Interfaces.PosPin

Electrical Pin (Positive)

Contents

Type	Name	Description
Voltage	v	[V]
flow Current	i	[A]

Modelica definition

```
connector PosPin "Electrical Pin (Positive)"
  Modelica.Units.SI.Voltage v;
  flow Modelica.Units.SI.Current i;
end PosPin;
```

EPowetrain.Interfaces.NegPin

Electrical Pin (Negative)

Contents

Type	Name	Description
Voltage	v	[V]
flow Current	i	[A]

Modelica definition

```
connector NegPin "Electrical Pin (Negative)"
  Modelica.Units.SI.Voltage v;
  flow Modelica.Units.SI.Current i;
end NegPin;
```

EPowetrain.Interfaces.ElectricPort

An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
partial model ElectricPort
  "An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-termina

  SI.Voltage v "Voltage between pines (= p.u - n.u)";
  flow SI.Current i "Current from pin p to pin n";
public
  PosPin p;
  NegPin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;

end ElectricPort;
```

EPowetrain.Interfaces.MechanicalAxis

Mechanical axis coupling

Contents

Type	Name	Description
Angle	Phi	[rad]
flow Torque	T	[N.m]

Modelica definition

```
connector MechanicalAxis
  "Mechanical axis coupling"
  SI.Angle Phi;
  flow SI.Torque T;
end MechanicalAxis;
```

EPowetrain.Interfaces.IO_Port

Input/output generalist port

Modelica definition

```
connector IO_Port = Real "Input/output generalist port";
```

EPowetrain.Interfaces.BoolOutPort

Boolean output

Modelica definition

```
connector BoolOutPort = output Boolean "Boolean output";
```

EPowetrain.Interfaces.BoolInPort

Boolean input

Modelica definition

```
connector BoolInPort = input Boolean "Boolean input";
```

EPowetrain.Interfaces.OutPort

Generalist output

Modelica definition

```
connector OutPort = output Real "Generalist output";
```

EPowetrain.Interfaces.InPort

Generalist input

Modelica definition

```
connector InPort = input Real "Generalist input";
```

EPowetrain.Interfaces.ThreePins

Triple electrical port used as base for transistors

Connectors

Type	Name	Description
PosPin	d	drain
PosPin	g	gate
PosPin	s	source

Modelica definition

```
partial model ThreePins
  "Triple electrical port used as base for transistors"

  EPowetrain.Interfaces.PosPin d "drain";
  EPowetrain.Interfaces.PosPin g "gate";
  EPowetrain.Interfaces.PosPin s "source";

end ThreePins;
```

EPowetrain.SignalRouting

Provides connectors, multiplexers, and demultiplexers for managing signal flow in large models or control systems.

Package Content

Name	Description
Not	Signal negation
Terminator	Signal terminator
unitDelay	Signal delay
Saturation	Signal saturation
AND	Boolean AND gate
Mux	Multiplexers
Multiply	Signal values multiplication

EPowetrain.SignalRouting.Not

Signal negation

Connectors

Type	Name	Description
input BoolInPort	In	
output BoolOutPort	Out	

Modelica definition

```
model Not "Signal negation"

  Interfaces.BoolInPort In;
  Interfaces.BoolOutPort Out;
equation
  Out = not (In);
end Not;
```

EPowetrain.SignalRouting.Terminator

Signal terminator

Connectors

Type	Name	Description
input InPort	inPort	

Modelica definition

```
model Terminator "Signal terminator"

  Interfaces.InPort inPort;
end Terminator;
```

EPowetrain.SignalRouting.unitDelay

Signal delay

Connectors

Type	Name	Description
input InPort	inPort	
output OutPort	outPort	

Modelica definition

```
model unitDelay "Signal delay"
  Interfaces.InPort inPort;
  Interfaces.OutPort outPort;
initial equation
  outPort = 0;
equation
  when time > 0 then
    outPort = pre(inPort);
  end when;

end unitDelay;
```

EPowetrain.SignalRouting.Saturation

Signal saturation

Parameters

Type	Name	Description
Real	Max	
Real	Min	

Connectors

Type	Name	Description
input InPort	In	
output OutPort	Out	

Modelica definition

```
model Saturation "Signal saturation"

  Interfaces.InPort In;
  Interfaces.OutPort Out;
  parameter Real Max=1e6;
  parameter Real Min=-1e-6;
equation
  Out = min(Max, max(Min, In));

end Saturation;
```

EPowetrain.SignalRouting.AND

Boolean AND gate

Connectors

Type	Name	Description
input BoolInPort	IN_1	
input BoolInPort	IN_2	
output BoolOutPort	Out	

Modelica definition

```
model AND "Boolean AND gate"
  Interfaces.BoolInPort IN_1;
  Interfaces.BoolInPort IN_2;
  Interfaces.BoolOutPort Out;
equation
  Out = if (IN_1 and IN_2) then true else false;

end AND;
```

EPowetrain.SignalRouting.Mux

Multiplexers

Package Content

Name	Description
Mux_3_in	Three signals to Array multiplexer
Mux_2_in	Two signals to Array multiplexer
Mux_2_in_Bool	Two boolean signals to Array multiplexer
ThreePhase_to_Bus	Three electrical signals to electrical bus
Bus_to_ThreePhase	Triple electrical bus to three signal demux
Demux_2	Two elements array to 2 signals
Demux_2_Bool	Two elements array to 2 boolean signals
Demux_3	Three elements array to 3 signals
Demux_6	Six elements array to 6 signals
BoolDemux_6	Six elements array to 6 boolean signals

EPowetrain.SignalRouting.Mux.Mux_3_in

Three signals to Array multiplexer

Connectors

Type	Name	Description
input InPort	In3	
input InPort	In2	
input InPort	In1	
output OutPort	out[3]	

Modelica definition

```
model Mux_3_in
  "Three signals to Array multiplexer"

  Interfaces.InPort In3;
  Interfaces.InPort In2;
  Interfaces.InPort In1;
  Interfaces.OutPort out[3];
equation
  out[1] = In1;
  out[2] = In2;
  out[3] = In3;

end Mux_3_in;
```

EPowetrain.SignalRouting.Mux.Mux_2_in

Two signals to Array multiplexer

Connectors

Type	Name	Description
input InPort	In2	
input InPort	In1	
output OutPort	out[2]	

Modelica definition

```
model Mux_2_in "Two signals to Array multiplexer"

  Interfaces.InPort In2;
  Interfaces.InPort In1;
  Interfaces.OutPort out[2];
equation
  out[1] = In1;
```

```
  out[2] = In2;

end Mux_2_in;
```

EPowetrain.SignalRouting.Mux.Mux_2_in_Bool

Two boolean signals to Array multiplexer

Connectors

Type	Name	Description
input BoolInPort	In2	
input BoolInPort	In1	
output BoolOutPort	out[2]	

Modelica definition

```
model Mux_2_in_Bool
  "Two boolean signals to Array multiplexer"

  Interfaces.BoolInPort In2;
  Interfaces.BoolInPort In1;
  Interfaces.BoolOutPort out[2];
equation
  out[1] = In1;
  out[2] = In2;

end Mux_2_in_Bool;
```

EPowetrain.SignalRouting.Mux.ThreePhase_to_Bus

Three electrical signals to electrical bus

Connectors

Type	Name	Description
NegPin	In3	
NegPin	In2	
NegPin	In1	
PosPin	out[3]	

Modelica definition

```
model ThreePhase_to_Bus
  "Three electrical signals to electrical bus"

  Interfaces.NegPin In3;
  Interfaces.NegPin In2;
  Interfaces.NegPin In1;
  Interfaces.PosPin out[3];
equation
  out[1] = In1;
  out[2] = In2;
  out[3] = In3;

end ThreePhase_to_Bus;
```

EPowetrain.SignalRouting.Mux.Bus_to_ThreePhase

Triple electrical bus to three signal demux

Connectors

Type	Name	Description
PosPin	out3	
PosPin	out1	

PosPin	out2	
NegPin	In[3]	

Modelica definition

```
model Bus_to_ThreePhase
  "Triple electrical bus to three signal demux"

  Interfaces.PosPin out3;
  Interfaces.PosPin out1;
  Interfaces.PosPin out2;
  Interfaces.NegPin In[3];
equation
  In[1] = out1;
  In[2] = out2;
  In[3] = out3;

end Bus_to_ThreePhase;
```

EPowetrain.SignalRouting.Mux.Demux_2

Two elements array to 2 signals

Connectors

Type	Name	Description
input InPort	In[2]	
output OutPort	out1	
output OutPort	out2	

Modelica definition

```
model Demux_2 "Two elements array to 2 signals"

  Interfaces.InPort In[2];
  Interfaces.OutPort out1;
  Interfaces.OutPort out2;
equation
  In[1] = out1;
  In[2] = out2;

end Demux_2;
```

EPowetrain.SignalRouting.Mux.Demux_2_Bool

Two elements array to 2 boolean signals

Connectors

Type	Name	Description
input BoolInPort	In[2]	
output BoolOutPort	out1	
output BoolOutPort	out2	

Modelica definition

```
model Demux_2_Bool
  "Two elements array to 2 boolean signals"

  Interfaces.BoolInPort In[2];
  Interfaces.BoolOutPort out1;
  Interfaces.BoolOutPort out2;
equation
  In[1] = out1;
  In[2] = out2;

end Demux_2_Bool;
```

EPowetrain.SignalRouting.Mux.Demux_3

Three elements array to 3 signals

Connectors

Type	Name	Description
input InPort	In[3]	
output OutPort	out1	
output OutPort	out3	
output OutPort	out2	

Modelica definition

```
model Demux_3 "Three elements array to 3 signals"

  Interfaces.InPort In[3];
  Interfaces.OutPort out1;
  Interfaces.OutPort out3;
  Interfaces.OutPort out2;
equation
  In[1] = out1;
  In[2] = out2;
  In[3] = out3;

end Demux_3;
```

EPowetrain.SignalRouting.Mux.Demux_6

Six elements array to 6 signals

Connectors

Type	Name	Description
input InPort	In[6]	
output OutPort	out1	
output OutPort	out3	
output OutPort	out2	
output OutPort	out4	
output OutPort	out5	
output OutPort	out6	

Modelica definition

```
model Demux_6 "Six elements array to 6 signals"

  Interfaces.InPort In[6];
  Interfaces.OutPort out1;
  Interfaces.OutPort out3;
  Interfaces.OutPort out2;
  Interfaces.OutPort out4;
  Interfaces.OutPort out5;
  Interfaces.OutPort out6;
equation
  In = {out1,out2,out3,out4,out5,out6};

end Demux_6;
```

EPowetrain.SignalRouting.Mux.BoolDemux_6

Six elements array to 6 boolean signals

Connectors

Type	Name	Description
input BoolInPort	In[6]	

output BoolOutPort	out1	
output BoolOutPort	out3	
output BoolOutPort	out2	
output BoolOutPort	out4	
output BoolOutPort	out5	
output BoolOutPort	out6	

Modelica definition

```
model BoolDemux_6
  "Six elements array to 6 boolean signals"

  Interfaces.BoolInPort In[6];
  Interfaces.BoolOutPort out1;
  Interfaces.BoolOutPort out3;
  Interfaces.BoolOutPort out2;
  Interfaces.BoolOutPort out4;
  Interfaces.BoolOutPort out5;
  Interfaces.BoolOutPort out6;
equation
  In = {out1,out2,out3,out4,out5,out6};

end BoolDemux_6;
```

EPowetrain.SignalRouting.Multiply

Signal values multiplication

Connectors

Type	Name	Description
output OutPort	Out	
input InPort	In1	
input InPort	In2	

Modelica definition

```
model Multiply "Signal values multiplication"
  Interfaces.OutPort Out;
  Interfaces.InPort In1;
  Interfaces.InPort In2;
equation
  Out = In1*In2;

end Multiply;
```

EPowetrain.Sources

Contains signal and power sources, including constant, step, sine and user-defined input profiles.

Package Content

Name	Description
Ramp	Step DC voltage source
Constant	Constant value
BoolConstant	Boolean constant
PWM	PWM voltage signal source
Sine	AC Voltage source
Step	Step DC voltage source
UDDS	Urban Dynamometer Driving Schedule (UDDS) drive cycle
BoolStep	Boolean step source
BoolPWM	PWM voltage signal source

EPowetrain.Sources.Ramp

Step DC voltage source

Parameters

Type	Name	Description
Time	St	[s]
Real	Slope	
Real	InitV	
Real	FinalV	

Connectors

Type	Name	Description
output OutPort	Out	

Modelica definition

```
model Ramp "Step DC voltage source"

  output Interfaces.OutPort Out;


  parameter Modelica.Units.SI.Time St=1;
  parameter Real Slope(min=0);
  parameter Real InitV=0;
  parameter Real FinalV=1;


initial equation

  Out = InitV;

equation

  if time >= St then
    if (InitV <= FinalV and Out <= FinalV) then
      der(Out) = Slope;
    elseif (InitV > FinalV and Out >= FinalV) then
      der(Out) = -Slope;
    else
      der(Out) = 0;
    end if;
  else
    der(Out) = 0;
  end if;

end Ramp;
```

EPowetrain.Sources.Constant

Constant value

Parameters

Type	Name	Description
Real	Value	

Connectors

Type	Name	Description
output OutPort	Out	

Modelica definition

```
model Constant "Constant value"

  parameter Real Value=0;
```

```
  EPowetrain.Interfaces.OutPort Out;
equation
  Out = Value;
end Constant;
```

EPowetrain.Sources.BoolConstant

Boolean constant

Parameters

Type	Name	Description
Boolean	Value	

Connectors

Type	Name	Description
output BoolOutPort	Out	

Modelica definition

```
model BoolConstant "Boolean constant"

  parameter Boolean Value=false;
  Interfaces.BoolOutPort Out;
equation
  Out = Value;
end BoolConstant;
```

EPowetrain.Sources.PWM

PWM voltage signal source

Parameters

Type	Name	Description
Real	D	Duty cicle
Real	Max	Maximun value
Real	Min	Minimum value
Time	time_start	start time [s]
Frequency	f	frequency [Hz]

Connectors

Type	Name	Description
output OutPort	Out	

Modelica definition

```
model PWM "PWM voltage signal source"

  parameter Real D(
    min=0,
    max=1) = 0.5 "Duty cicle";
  parameter Real Max=1 "Maximun value";
  parameter Real Min=0 "Minimum value";
  parameter SI.Time time_start(min=0) = 0 "start time";
  parameter SI.Frequency f=1 "frequency";
  SI.Time t(start=0);
  SI.Time T=1/f;
  SI.Time TOn=D*T;

  constant Real K=1e5;

  Interfaces.OutPort Out;
equation

  der(t) = 1;
```

```
  when (t >= T) then
    reinit(t, 0);
  end when;

  der(Out) = if (time >= time_start and t <= TOn) then
    K*(Max - Out) else K*(Min - Out);

end PWM;
```

EPowetrain.Sources.Sine

AC Voltage source

Parameters

Type	Name	Description
Voltage	U0	Amplitude [V]
Frequency	f	Frequency [Hz]
Angle	phi	Phase shift [rad]

Connectors

Type	Name	Description
output OutPort	p	

Modelica definition

```
model Sine "AC Voltage source"

  parameter SI.Voltage U0=1 "Amplitude";
  parameter SI.Frequency f=50 "Frequency";
  parameter SI.Angle phi=0 "Phase shift";

protected
  parameter Modelica.Units.SI.AngularFrequency w=2*
      Modelica.Constants.pi*f;
public
  Interfaces.OutPort p;
equation

  p = U0*sin(w*time + phi);
end Sine;
```

EPowetrain.Sources.Step

Step DC voltage source

Parameters

Type	Name	Description
replaceable type T		
Time	St	[s]
Real	InitV	
Real	FinalV	

Connectors

Type	Name	Description
output OutPort	Out	
replaceable type T		

Modelica definition

```
model Step "Step DC voltage source"

  output Interfaces.OutPort Out;
```

```
  replaceable type T = Real;
  parameter Modelica.Units.SI.Time St=1;
  parameter T InitV=0;
  parameter T FinalV=1;

equation
  if time >= St then
    Out = FinalV;
  else
    Out = InitV;
  end if;

end Step;
```

EPowetrain.Sources.UDDS

Urban Dynamometer Driving Schedule (UDDS) drive cycle

Information

Extends from Modelica.Blocks.Sources.TimeTable (Generate a (possibly discontinuous) signal by linear interpolation in a table).

Parameters

Type	Name	Description
Real	table[:, 2]	Table matrix (time = first column; e.g., table=[0, 0; 1, 1; 2, 4])
Time	timeScale	Time scale of first table column [s]
Real	offset	Offset of output signal y
Time	startTime	Output y = offset for time < startTime [s]
Time	shiftTime	Shift time of first table column [s]

Connectors

Type	Name	Description
output RealOutput	y	Connector of Real output signal

Modelica definition

```
model UDDS
  "Urban Dynamometer Driving Schedule (UDDS) drive cycle"
  extends Modelica.Blocks.Sources.TimeTable(table=fill(0.0,
        0, 2));
initial equation

end UDDS;
```

EPowetrain.Sources.BoolStep

Boolean step source

Parameters

Type	Name	Description
Time	St	[s]
Boolean	InitV	
Boolean	FinalV	

Connectors

Type	Name	Description
output BoolOutPort	Out	

Modelica definition

```
model BoolStep "Boolean step  source"

  output Interfaces.BoolOutPort Out;

  parameter Modelica.Units.SI.Time St=1;
  parameter Boolean InitV=false;
  parameter Boolean FinalV=true;

equation
  if time >= St then
    Out = FinalV;
  else
    Out = InitV;
  end if;

end BoolStep;
```

EPowetrain.Sources.BoolPWM

PWM voltage signal source

Parameters

Type	Name	Description
Real	D	Duty cicle
Time	time_start	start time [s]
Frequency	f	frequency [Hz]

Connectors

Type	Name	Description
output BoolOutPort	Out	

Modelica definition

```
model BoolPWM "PWM voltage signal source"

  parameter Real D(
    min=0,
    max=1) = 0.5 "Duty cicle";
  parameter SI.Time time_start(min=0) = 0 "start time";
  parameter SI.Frequency f=1 "frequency";
  SI.Time t(start=0);
  SI.Time T=1/f;
  SI.Time TOn=D*T;

  Interfaces.BoolOutPort Out;
equation

  der(t) = 1;

  Out = (time >= time_start and t <= TOn);

  when (t >= T) then
    reinit(t, 0);
  end when;
end BoolPWM;
```

EPowetrain.Electrical

Includes electrical components such as resistors, capacitors, inductors, diodes, and configurable sources. These are used as building blocks for more complex models.

Package Content

Name	Description
Sources	Electrical sources
Basic	Basic, electrical components
Semiconductors	

Devices	Device models such as electric motors, converters, and battery packs

EPowetrain.Electrical.Sources

Electrical sources

Package Content

Name	Description
AC_Source	AC Voltage source
DC_Source	DC voltage source
DC_Current_Source	Direct current source
VStep	Step DC voltage source
Battery	DC voltage source
Var_DC_Source	Variable DC voltage source
Ground	Zero voltage reference
Square	Squarevoltage signal source

EPowetrain.Electrical.Sources.AC_Source

AC Voltage source

Parameters

Type	Name	Description
Voltage	U0	Amplitude [V]
Frequency	f	Frequency [Hz]
Angle	phi	Phase shift [rad]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model AC_Source "AC Voltage source"

  parameter SI.Voltage U0=1 "Amplitude";
  parameter SI.Frequency f=50 "Frequency";
  parameter SI.Angle phi=0 "Phase shift";
  SI.Angle theta;

  constant Real pi=Modelica.Constants.pi;

  SI.Voltage v;
  SI.Current i;


public
  EPowetrain.Interfaces.PosPin p;
  EPowetrain.Interfaces.NegPin n;
initial equation
  theta = phi;

equation
  0 = p.i + n.i;
  i = p.i;
  v = p.v - n.v;

  when theta >= 2*pi then
    reinit(theta, 0);
  end when;

  der(theta) = 2*pi*f;
```

```
  v = U0*Modelica.Math.sin(theta);
end AC_Source;
```

EPowetrain.Electrical.Sources.DC_Source

DC voltage source

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Voltage	U0	[V]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model DC_Source "DC voltage source"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Voltage U0=12;
equation
  v = U0;
end DC_Source;
```

EPowetrain.Electrical.Sources.DC_Current_Source

Direct current source

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Current	I0	[A]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model DC_Current_Source
  "Direct current source"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Current I0=1;
equation
  i = I0;
end DC_Current_Source;
```

EPowetrain.Electrical.Sources.VStep

Step DC voltage source

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Time	St	[s]
Real	v0	
Real	vf	

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model VStep "Step DC voltage source"
  extends EPowetrain.Interfaces.ElectricPort;

  Real v_step(start=v0);

  parameter Modelica.Units.SI.Time St=1;
  parameter Real v0=0;
  parameter Real vf=1;
equation
  v_step = if time < St then v0 else vf;
  v = v_step;
end VStep;
```

EPowetrain.Electrical.Sources.Battery

DC voltage source

Parameters

Type	Name	Description
ElectricCharge_Ah	Cap	Battery capacity [A.h]
Voltage	Vd	Dischargued voltage (SOC = 0) [V]
Voltage	Vf	Full voltage (SOC = 100) [V]
Real	InitSOC	Initial state of charge [1]
Resistance	Rs	Serie resistance [Ohm]
Resistance	Rp	Parallel resistance [Ohm]
Capacitance	C	Battery capacitance [F]
Current	Imax	Maximum, peak current [A]

Connectors

Type	Name	Description
PosPin	posPin	
NegPin	negPin	

Modelica definition

```
model Battery "DC voltage source"

  parameter Modelica.Units.NonSI.ElectricCharge_Ah
```

```
    Cap(min=0) = 1 "Battery capacity";

  parameter Modelica.Units.SI.Voltage Vd=0
    "Dischargued voltage (SOC = 0)";
  parameter Modelica.Units.SI.Voltage Vf=12
    "Full voltage (SOC = 100)";

  parameter Real InitSOC(
    quantity="Percent",
    final unit="1",
    final displayUnit="%",
    min=0,
    max=100) = 100 "Initial state of charge";

  Real SOC(
    quantity="Percent",
    final unit="1",
    final displayUnit="%",
    min=0,
    max=100);

  parameter SI.Resistance Rs=0.06 "Serie resistance";
  parameter SI.Resistance Rp=1e-3
    "Parallel resistance";
  parameter SI.Capacitance C=1e-6
    "Battery capacitance";
  parameter SI.Current Imax=400
    "Maximum, peak current";

  SI.Voltage Vout=posPin.v - negPin.v;
  SI.Current Iout(
    min=-Imax,
    max=Imax) = posPin.i;



  Interfaces.PosPin posPin;
  Interfaces.NegPin negPin;

  Basic.Resistance RP(R=Rp);
public
  Basic.Capacitor C1(C=C);
  Basic.Resistance RS(R=Rs);
public
  Var_DC_Source Batt;
  Devices.CurrentSaturation currentSaturation(IMax=
        Imax, IMin=-Imax);
initial equation
  SOC = InitSOC;
  RS.v = 0;
  C1.v = 0;
equation
  der(SOC) = Iout/(3600*Cap);

  if SOC > 0 then
    Batt.v = Vd + (Vf - Vd)*SOC/100;
  else
    Batt.i = 0;
  end if;



  connect(C1.p, RS.n);
  connect(RP.p, RS.n);
  connect(C1.n, RP.n);
  connect(Batt.p, RP.n);
  connect(Batt.n, negPin);
  connect(currentSaturation.In, RS.p);
  connect(currentSaturation.Out, posPin);


end Battery;
```

EPowetrain.Electrical.Sources.Var_DC_Source

Variable DC voltage source

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model Var_DC_Source
  "Variable DC voltage source"

  SI.Voltage v "Voltage between pines (= p.u - n.u)";
  flow SI.Current i "Current from pin p to pin n";

  Interfaces.PosPin p;
  Interfaces.NegPin n;
equation

  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Var_DC_Source;
```

EPowetrain.Electrical.Sources.Ground

Zero voltage reference

Connectors

Type	Name	Description
PosPin	p	

Modelica definition

```
model Ground "Zero voltage reference"

  EPowetrain.Interfaces.PosPin p;
equation
  p.v = 0;
end Ground;
```

EPowetrain.Electrical.Sources.Square

Squarevoltage signal source

Parameters

Type	Name	Description
Real	D	Duty cicle
Time	time_start	start time [s]
Frequency	f	frequency [Hz]
Voltage	Von	[V]
Voltage	Voff	[V]
Real	k	smooth factor

Connectors

Type	Name	Description
PosPin	Out	

Modelica definition

```
model Square "Squarevoltage signal source"

  Interfaces.PosPin Out;

  parameter Real D(
    min=0,
    max=1) = 0.5 "Duty cicle";
```

```
  parameter SI.Time time_start(min=0) = 0
    "start time";
  parameter SI.Frequency f=1 "frequency";
  parameter SI.Voltage Von=1;
  parameter SI.Voltage Voff=0;
  parameter Real k=1000 "smooth factor";

  SI.Time t(start=0);
  SI.Time T=1/f;
  SI.Time eps=1e-6;
  SI.Time TOn=D*T;

equation

  der(t) = 1;

  when (t >= T) then
    reinit(t, 0);
  end when;

  der(Out.v) = if (t < TOn) then k*(Von - Out.v)
     else k*(Voff - Out.v);


end Square;
```

EPowetrain.Electrical.Basic

Basic, electrical components

Package Content

Name	Description
IdealCoil	Ideal coil
Resistance	Ideal resistance
VariableResistance	Variable parameter resistance
Capacitor	Ideal capacitor
VariableCapacitor	Variable parameter capacitor

EPowetrain.Electrical.Basic.IdealCoil

Ideal coil

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Inductance	L	Inductance [H]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model IdealCoil "Ideal coil"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Inductance L=1
    "Inductance";
equation
  v = smooth(1, L*der(i));
```

```
end IdealCoil;
```

EPowetrain.Electrical.Basic.Resistance

Ideal resistance

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Resistance	R	Resistance [Ohm]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model Resistance "Ideal resistance"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Resistance R=100
    "Resistance";
equation
  v = smooth(1, R*i);
end Resistance;
```

EPowetrain.Electrical.Basic.VariableResistance

Variable parameter resistance

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model VariableResistance
  "Variable parameter resistance"
  extends EPowetrain.Interfaces.ElectricPort;
  Modelica.Units.SI.Resistance R "Resistance";
equation
  v = smooth(1, R*i);
end VariableResistance;
```

EPowetrain.Electrical.Basic.Capacitor

Ideal capacitor

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Capacitance	C	Capacitance [F]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model Capacitor "Ideal capacitor"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Capacitance C=1e-6
    "Capacitance";
equation
  i = smooth(1, C*der(v));

end Capacitor;
```

EPowetrain.Electrical.Basic.VariableCapacitor

Variable parameter capacitor

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model VariableCapacitor
  "Variable parameter capacitor"
  extends EPowetrain.Interfaces.ElectricPort;
  Modelica.Units.SI.Capacitance C "Capacitance";
equation
  i = smooth(1, C*der(v));

end VariableCapacitor;
```

EPowetrain.Electrical.Semiconductors

Package Content

Name	Description
NMOS	Ideal NMOS
PMOS	Ideal PMOS
Diode	Ideal diode
IdealIGBT	Ideal Insulated Gate Bipolar Transistor (IGBT)
IdealISwitch	Ideal electrical switch

EPowetrain.Electrical.Semiconductors.NMOS

Ideal NMOS

Parameters

Type	Name	Description
Voltage	Vt	[V]
Length	L	[m]
Length	W	[m]
Real	Kp	[A/V^2]
Real	Lambda	[V^-1]

Connectors

Type	Name	Description
PosPin	d	drain
PosPin	g	gate
PosPin	s	source

Modelica definition

```
model NMOS "Ideal NMOS"

  EPowetrain.Interfaces.PosPin d "drain";
  EPowetrain.Interfaces.PosPin g "gate";
  EPowetrain.Interfaces.PosPin s "source";

  Modelica.Units.SI.Current Ids(start=0);
  Modelica.Units.SI.Voltage Vgs(start=0)
    "Pins voltage (= g.v - s.v)";
  Modelica.Units.SI.Voltage Vds(start=0)
    "Pins voltage  (= d.v - s.v)";

  parameter Modelica.Units.SI.Voltage Vt=3;
  parameter Modelica.Units.SI.Length L=2e-6;
  parameter Modelica.Units.SI.Length W=10e-6;
  parameter Real Kp(unit="A/V^2") = 100e-6;
  parameter Real Lambda(unit="V^-1") = 0;

equation
  Vgs = g.v - s.v;
  Vds = d.v - s.v;

  g.i = 0;
  d.i = Ids;
  d.i = -s.i;

  // Cut
  if (Vgs < Vt) then
    Ids = 0;
  else
    // Lineal
    if (Vds <= (Vgs - Vt)) then
      Ids = Kp*(W/L)*((Vgs - Vt)*Vds - (Vds^2)/2)*(1 +
        Lambda*Vds);
      // Saturation
    else
      Ids = 0.5*Kp*(W/L)*(Vgs - Vt)^2*(1 + Lambda*Vds);
    end if;
  end if;

end NMOS;
```

EPowetrain.Electrical.Semiconductors.PMOS

Ideal PMOS

Parameters

Type	Name	Description
Voltage	Vt	[V]
Length	L	[m]
Length	W	[m]
Real	Kp	[A/V^2]
Real	Lambda	[V^-1]

Connectors

Type	Name	Description
PosPin	d	source
PosPin	g	gate
PosPin	s	drain

Modelica definition

```
model PMOS "Ideal PMOS"

  EPowetrain.Interfaces.PosPin d "source";
  EPowetrain.Interfaces.PosPin g "gate";
  EPowetrain.Interfaces.PosPin s "drain";

  Modelica.Units.SI.Current Ids(start=0);
  Modelica.Units.SI.Voltage Vgs(start=0)
    "Pins voltage (= g.v - s.v)";
  Modelica.Units.SI.Voltage Vds(start=0)
    "Pins voltage (= d.v - s.v)";

  parameter Modelica.Units.SI.Voltage Vt=3;
  parameter Modelica.Units.SI.Length L=2e-6;
  parameter Modelica.Units.SI.Length W=10e-6;
  parameter Real Kp(unit="A/V^2") = 100e-6;
  parameter Real Lambda(unit="V^-1") = 0;

equation
  Vgs = g.v - s.v;
  Vds = d.v - s.v;

  g.i = 0;
  d.i = Ids;
  d.i = -s.i;

  // Cut
  if (Vgs > Vt) then
    Ids = 0;
  else
    // Lineal
    if (Vds >= (Vgs - Vt)) then
      Ids = Kp*(W/L)*((Vgs - Vt)*Vds - (Vds^2)/2)*(1 +
        Lambda*Vds);
      // Saturation
    else
      Ids = 0.5*Kp*(W/L)*(Vgs - Vt)^2*(1 + Lambda*Vds);
    end if;
  end if;

end PMOS;
```

EPowetrain.Electrical.Semiconductors.Diode

Ideal diode

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description

Current	Is	Saturation current [A]
Voltage	Vt	Thermal voltage [V]
Real	Maxexp	Max. exponent for linear continuation
Resistance	R	Parallel ohmic resistance [Ohm]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	

Modelica definition

```
model Diode "Ideal diode"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter Modelica.Units.SI.Current Is=1e-6
    "Saturation current";
  parameter Modelica.Units.SI.Voltage Vt=0.04
    "Thermal voltage";
  parameter Real Maxexp(final min=Modelica.Constants.small)
     = 15 "Max. exponent for linear continuation";
  parameter SI.Resistance R=1.e8
    "Parallel ohmic resistance";
equation

  i = smooth(1, (if (v/Vt > Maxexp) then Is*(exp(
    Maxexp)*(1 + v/Vt - Maxexp) - 1) + v/R else Is*(
    exp(v/Vt) - 1) + v/R));
end Diode;
```

EPowetrain.Electrical.Semiconductors.IdealIGBT

Ideal Insulated Gate Bipolar Transistor (IGBT)

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Current	Is	Saturation current [A]
Voltage	Vt	Thermal voltage [V]
Real	Maxexp	Max. exponent for linear continuation
Resistance	R	Parallel ohmic resistance [Ohm]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	
PosPin	c	

Modelica definition

```
model IdealIGBT
  "Ideal Insulated Gate Bipolar Transistor (IGBT)"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter SI.Current Is=1e-9 "Saturation current";
  parameter SI.Voltage Vt=0.025 "Thermal voltage";
  parameter Real Maxexp(final min=Modelica.Constants.small)
     = 15 "Max. exponent for linear continuation";
  parameter SI.Resistance R=1.e8
    "Parallel ohmic resistance";
  EPowetrain.Interfaces.PosPin c;
equation
```

```
  c.i = 0;
  if (c.v > 0) then
    i = smooth(1, (if (v/Vt > Maxexp) then Is*(exp(
      Maxexp)*(1 + v/Vt - Maxexp) - 1) + v/R else Is*(
      exp(v/Vt) - 1) + v/R));
  else
    i = smooth(1, v/R);
  end if;

end IdealIGBT;
```

EPowetrain.Electrical.Semiconductors.IdealISwitch

Ideal electrical switch

Information

Extends from EPowetrain.Interfaces.ElectricPort (An abstract base model that groups a PosPin and a NegPin. Used as a base class for all two-terminal electrical components, such as resistors, capacitors, sources, and converters.).

Parameters

Type	Name	Description
Resistance	ROpen	Opened circuit conductance [Ohm]
Resistance	RClose	Closed circuit resistance [Ohm]
Time	St	Switch time [s]

Connectors

Type	Name	Description
PosPin	p	
NegPin	n	
input BoolInPort	c	

Modelica definition

```
model IdealISwitch "Ideal electrical switch"
  extends EPowetrain.Interfaces.ElectricPort;
  parameter SI.Resistance ROpen=1e5
    "Opened circuit conductance";
  parameter SI.Resistance RClose=1.e-5
    "Closed circuit resistance";
  parameter SI.Time St(min=1e-9) = 1e-6 "Switch time";

  Real Smooth_aux
    "Aux variable to smooth resistance variation";

  SI.Resistance R(start=ROpen);


  Interfaces.BoolInPort c;


equation

  der(Smooth_aux)*St = if c then 1 - Smooth_aux else -
    Smooth_aux;

  R = ROpen + (RClose - ROpen)*Smooth_aux;

  i = v/R;


end IdealISwitch;
```

EPowetrain.Electrical.Devices

Device models such as electric motors, converters, and battery packs

Package Content

Name	Description
Converters	Power converters
Machines	Electrical machines
CurrentSaturation	Current limitation

EPowetrain.Electrical.Devices.Converters

Power converters

Package Content

Name	Description
BackEMF	Counter-electromotive force
ElectricConverter	DC/DC ideal power converter

EPowetrain.Electrical.Devices.Converters.BackEMF

Counter-electromotive force

Parameters

Type	Name	Description
Real	Ke	Back emf constant [V.s/rad]
Real	Kt	Torque constant [N.m/A]
RotationalDampingConstant	bm	Friction constant [N.m.s/rad]
Inertia	J	[kg.m2]

Connectors

Type	Name	Description
PosPin	Pp	
NegPin	Np	
MechanicalAxis	mechanicalAxis	

Modelica definition

```
model BackEMF
  "Counter-electromotive force"
  Interfaces.PosPin Pp;
  Interfaces.NegPin Np;
  Interfaces.MechanicalAxis mechanicalAxis;
  parameter Real Ke(unit="V.s/rad") = 0.0064
    "Back emf constant";
  parameter Real Kt(unit="N.m/A") = 0.0065
    "Torque constant";
  parameter SI.RotationalDampingConstant bm=4.121*1e-6
    "Friction constant";
  parameter Modelica.Units.SI.Inertia J=3.87*1e-7;
  Modelica.Units.SI.Voltage Vemf;
  Modelica.Units.SI.Current i=Pp.i;
  Modelica.Units.SI.Torque Te "Electrical torque";
  Modelica.Units.SI.Torque Tb "Friction torque";
  Modelica.Units.SI.Torque Tload=mechanicalAxis.T
    "Axis torque";
  Modelica.Units.SI.Angle Phi=mechanicalAxis.Phi;
  Modelica.Units.SI.AngularVelocity w=der(Phi);
  constant Real pi=Modelica.Constants.pi;

equation
  Pp.i + Np.i = 0;
  Vemf = Ke*w;
  Vemf = Pp.v - Np.v;
  Te = i*Kt;
  Tb = bm*w;
```

```
  Te - Tb - Tload = J*der(w);


end BackEMF;
```

EPowetrain.Electrical.Devices.Converters.ElectricConverter

DC/DC ideal power converter

Connectors

Type	Name	Description
PosPin	P_In	
NegPin	N_In	
PosPin	P_Out	
NegPin	N_Out	
input InPort	DutyCycle	

Modelica definition

```
model ElectricConverter
  "DC/DC ideal power converter"
  Interfaces.PosPin P_In;
  Interfaces.NegPin N_In;
  Interfaces.PosPin P_Out;
  Interfaces.NegPin N_Out;
  Interfaces.InPort DutyCycle;
  input SI.Voltage V_In=P_In.v - N_In.v;
  output SI.Voltage V_Out=P_Out.v - N_Out.v;
  input SI.Current I_In;
  // Input port current (Source side)
  output SI.Current I_Out=P_Out.i;
  // Output port curren (Load side)

  SI.Power PwIn=V_In*I_In;
  SI.Power PwOut=V_Out*I_Out;

  SI.Energy EBalance(start=0);
equation

  der(EBalance) = PwIn + PwOut;

  I_In = if DutyCycle >= 0 then P_In.i else -P_In.i;

  // Set V_Out
  V_Out = DutyCycle*V_In;

  PwIn + PwOut = 0;

  //Currents balnce
  P_In.i + N_In.i = 0;
  P_Out.i + N_Out.i = 0;


end ElectricConverter;
```

EPowetrain.Electrical.Devices.Machines

Electrical machines

Package Content

Name	Description
PMSM	Permanent magnet synchronous motor
DCMotor	Permanent magnet DC motor

EPowetrain.Electrical.Devices.Machines.PMSM

Permanent magnet synchronous motor

Parameters

Type	Name	Description
Integer	Pp	Poles pairs
Resistance	Rs	Stator winding resistance [Ohm]
Inductance	Ld	[H]
Inductance	Lq	[H]
Inertia	J	[kg.m2]
Real	Bv	Dynamic viscosity [N.m.s/rad]
MagneticFlux	Fmg	[Wb]

Connectors

Type	Name	Description
PosPin	Vin[3]	
MechanicalAxis	Rotor	

Modelica definition

```
model PMSM "Permanent magnet synchronous motor"

  constant Real pi=Constant.pi;
  Interfaces.PosPin Vin[3];
  Interfaces.MechanicalAxis Rotor;
  parameter Integer Pp(min=1) = 2 "Poles pairs";
  parameter SI.Resistance Rs=2.98
    "Stator winding resistance";
  parameter SI.Inductance Ld=7e-3;
  parameter SI.Inductance Lq=7e-3;
  parameter SI.Inertia J=4.7e-5;
  parameter Real Bv(
    unit="N.m.s/rad",
    min=0) = 1.1e-4 " Dynamic viscosity";
  parameter SI.MagneticFlux Fmg=0.125;
  SI.Angle Th_e;
  SI.Angle Th_m;
  SI.AngularVelocity we
    "Electrical angular velocity";
  SI.AngularVelocity wm(start=0)
    "Mechanical angular velocity";
  SI.Torque Tl=Rotor.T;
  SI.Torque Te;
  SI.Current Ia=Vin[1].i;
  SI.Current Ib=Vin[2].i;
  SI.Current Ic=Vin[3].i;
  SI.Current Id "Current on direct axis";
  SI.Current Iq "Current on normal axis";
  SI.Voltage Va=smooth(1, Vin[1].v);
  SI.Voltage Vb=smooth(1, Vin[2].v);
  SI.Voltage Vc=smooth(1, Vin[3].v);
  SI.Voltage Vd;
  SI.Voltage Vq;
  SI.MagneticFlux Fd "Stator magnetic fluxes";
  SI.MagneticFlux Fq
    "Stator permanent magnetic fluxes";
  Real PT[2,3]=2/3*[cos(Th_e),cos(Th_e - 2*pi/3),
      cos(Th_e - 4*pi/3); -sin(Th_e),-sin(Th_e - 2*
      pi/3),-sin(Th_e - 4*pi/3)];
equation
  [Id; Iq] = PT*[Ia; Ib; Ic];
  [Vd; Vq] = PT*[Va; Vb; Vc];
  Ia + Ib + Ic = 0;

  Fd = Ld*Id + Fmg;
  Fq = Lq*Iq;

  Vq = Rs*Iq + we*Fd + der(Fq);
  Vd = Rs*Id - we*Fq + der(Fd);

  Te = 1.5*Pp*(Fd*Iq - Fq*Id);
```

```
  Te - Tl - Bv*wm = J*der(wm);


  we = Pp*wm;

  der(Th_e) = we;
  der(Th_m) = wm;

  Rotor.Phi = Th_m;


end PMSM;
```

EPowetrain.Electrical.Devices.Machines.DCMotor

Permanent magnet DC motor

Parameters

Type	Name	Description
Resistance	Rm	Motor electrical resistance [Ohm]
Inductance	Lm	Motor electrical inductance [H]
Real	Ke	Back emf constant [V.s/rad]
Real	Kt	Torque constant [N.m/A]
RotationalDampingConstant	bm	Visc. friction constant [N.m.s/rad]
Inertia	J	Rotor's inertia [kg.m2]

Connectors

Type	Name	Description
PosPin	Vp	
MechanicalAxis	Rotor	
NegPin	Vn	

Modelica definition

```
model DCMotor "Permanent magnet DC motor"

  constant Real pi=Constant.pi;
  Interfaces.PosPin Vp;
  Interfaces.MechanicalAxis Rotor;



  Interfaces.NegPin Vn;
  Basic.Resistance R1(R=Rm);
  Basic.IdealCoil L1(L=Lm);


  parameter Modelica.Units.SI.Resistance Rm=0.837
    "Motor electrical resistance";
  parameter Modelica.Units.SI.Inductance Lm=0.0008
    "Motor electrical inductance";

  parameter Real Ke(unit="V.s/rad") = 0.0064
    "Back emf constant";
  parameter Real Kt(unit="N.m/A") = 0.0065
    "Torque constant";
  parameter Modelica.Units.SI.RotationalDampingConstant
                                                  bm=4.121
      *1e-6 "Visc. friction constant";
  parameter Modelica.Units.SI.Inertia J=3.87*1e-7
    "Rotor's inertia";

  Converters.BackEMF backEMF(
    Ke=Ke,
    Kt=Kt,
    bm=bm,
    J=J);
equation
  connect(backEMF.mechanicalAxis, Rotor);
```

```
  connect(R1.p, L1.n);
  connect(R1.n, Vp);
  connect(L1.p, backEMF.Pp);
  connect(backEMF.Np, Vn);

end DCMotor;
```

EPowetrain.Electrical.Devices.CurrentSaturation

Current limitation

Parameters

Type	Name	Description
Real	IMax	
Real	IMin	

Connectors

Type	Name	Description
input NegPin	In	
output PosPin	Out	

Modelica definition

```
model CurrentSaturation "Current limitation"

  input Interfaces.NegPin In;
  output Interfaces.PosPin Out;
  parameter Real IMax=1e6;
  parameter Real IMin=-1e-6;

  SI.Current Iout=Out.i;
  SI.Current Iin=In.i;
equation
  Iout = min(IMax, max(IMin, Iin));
  Out.v = In.v;

end CurrentSaturation;
```

EPowetrain.Mechanical

Provides mechanical elements for translational and rotational motion, including inertia, load models, and coupling flanges for drivetrain representation.

Package Content

Name	Description
Basic	
BodyFrame1DOF	1 degree of freedom body frame
Slope	Terrain Slope

EPowetrain.Mechanical.Basic

Package Content

Name	Description
RotLoad	Rotational Load
RotAxis	Rotational Axis connector
Wheel	Wheel load model
FixedTorque	Constant torque source
FixedSpeed	Constant angular speed source
Gearbox	Toque and ang. Velocity converter

EPowetrain.Mechanical.Basic.RotLoad

Rotational Load

Parameters

Type	Name	Description
Inertia	J	Inertial load [kg.m2]
DynamicViscosity	fs	Dynamic viscosity [Pa.s]

Connectors

Type	Name	Description
MechanicalAxis	Axis	

Modelica definition

```
model RotLoad "Rotational Load"

  Interfaces.MechanicalAxis Axis;
  parameter SI.Inertia J=10 "Inertial load";
  parameter SI.DynamicViscosity fs=8.5e-6
    " Dynamic viscosity";
  constant Real eps=1e-3;
  SI.AngularVelocity w;
equation
  der(Axis.Phi) = w;
  J*der(w) = -Axis.T - w*fs;

end RotLoad;
```

EPowetrain.Mechanical.Basic.RotAxis

Rotational Axis connector

Parameters

Type	Name	Description
Inertia	J	Inertial load [kg.m2]

Connectors

Type	Name	Description
MechanicalAxis	AxisA	
MechanicalAxis	AxisB	

Modelica definition

```
model RotAxis "Rotational Axis connector"

  Interfaces.MechanicalAxis AxisA;
  Interfaces.MechanicalAxis AxisB;
  parameter SI.Inertia J=10 "Inertial load";
  SI.AngularVelocity w;
  SI.Angle Phi;
equation
  der(Phi) = w;
  J*der(w) = AxisA.T + AxisB.T;
  AxisA.Phi = Phi;
  AxisB.Phi = Phi;


end RotAxis;
```

EPowetrain.Mechanical.Basic.Wheel

Wheel load model

Parameters

Type	Name	Description
Inertia	J	Inertial load [kg.m2]
Length	R	Radius [m]
Real	fs	Dynamic viscosity [N.m.s/rad]
Mass	M	Vehicle mass [kg]

Connectors

Type	Name	Description
MechanicalAxis	Axis	
output OutPort	Vwheel	

Modelica definition

```
model Wheel "Wheel load model"
  constant Real pi=Constant.pi;
  Interfaces.MechanicalAxis Axis;
  parameter SI.Inertia J=10 "Inertial load";
  parameter SI.Length R=0.3 "Radius";
  parameter Real fs(
    unit="N.m.s/rad",
    min=0) = 8.5e-6 " Dynamic viscosity";
  parameter SI.Mass M=1200 "Vehicle mass";
  constant Real eps=1e-3;
  SI.AngularVelocity w;
  SI.Velocity V(start=0);
  Interfaces.OutPort Vwheel;
equation
  der(Axis.Phi) = w;

  V = 2*pi*R*w;

  if abs(V) < eps and abs(Axis.T) < eps then
    //Stacionary state
    der(w) = -1e3*w;
  else
    Axis.T + w*fs + (J + M*R^2)*der(w) = 0;

  end if;
  Vwheel = V;

end Wheel;
```

EPowetrain.Mechanical.Basic.FixedTorque

Constant torque source

Parameters

Type	Name	Description
Torque	T	Torque [N.m]

Connectors

Type	Name	Description
MechanicalAxis	Axis	

Modelica definition

```
model FixedTorque "Constant torque source"

  Interfaces.MechanicalAxis Axis;
  parameter SI.Torque T=10 "Torque";

equation
```

```
  T = Axis.T;

end FixedTorque;
```

EPowetrain.Mechanical.Basic.FixedSpeed

Constant angular speed source

Parameters

Type	Name	Description
AngularVelocity	w	Angular velocity [rad/s]

Connectors

Type	Name	Description
MechanicalAxis	Axis	

Modelica definition

```
model FixedSpeed "Constant angular speed source"

  Interfaces.MechanicalAxis Axis;
  parameter SI.AngularVelocity w=1 "Angular velocity";
equation
  der(Axis.Phi) = w;


end FixedSpeed;
```

EPowetrain.Mechanical.Basic.Gearbox

Toque and ang. Velocity converter

Parameters

Type	Name	Description
Real	N	Conversion ratio

Connectors

Type	Name	Description
input MechanicalAxis	AxisA	
output MechanicalAxis	AxisB	

Modelica definition

```
model Gearbox "Toque and ang. Velocity converter"

  input Interfaces.MechanicalAxis AxisA;
  output Interfaces.MechanicalAxis AxisB;
  parameter Real N(min=0.01) = 1 "Conversion ratio";

  SI.Torque TIn=AxisA.T;
  SI.Torque TOut=AxisB.T;


  SI.Angle PhiIn=AxisA.Phi;
  SI.Angle PhiOut=AxisB.Phi;
equation

  der(PhiOut) = der(N*PhiIn);
  TIn + N*TOut = 0;


end Gearbox;
```

EPowetrain.Mechanical.BodyFrame1DOF

1 degree of freedom body frame

Parameters

Type	Name	Description
Length	R	Wheel radius [m]
Mass	M	Vehicle mass [kg]
Area	Af	Vehicle front area [m2]
Real	Cd	Vehicle drag coef.
Real	Cr	Tyres rolling resistance coef.
Density	rho	Air density [kg/m3]

Connectors

Type	Name	Description
MechanicalAxis	TorqueIN	
output OutPort	V	Vehicle speed
input InPort	Alpha	Terrain slope

Modelica definition

```
model BodyFrame1DOF
  "1 degree of freedom body frame"

  Interfaces.MechanicalAxis TorqueIN;
  output Interfaces.OutPort V "Vehicle speed";
  input Interfaces.InPort Alpha "Terrain slope";
  constant SI.Acceleration g=Modelica.Constants.g_n;
  constant Real pi=Modelica.Constants.pi;
  parameter SI.Length R(min=0.01) = 0.25 "Wheel radius";
  parameter SI.Mass M=1500 "Vehicle mass";
  parameter SI.Area Af=2 "Vehicle front area";
  parameter Real Cd(min=0) = 0.248 "Vehicle drag coef.";
  parameter Real Cr(min=0) = 0.01
    "Tyres rolling resistance coef.";
  parameter SI.Density rho(displayUnit="kg/m3") = 1.2
    "Air density";

  SI.Force F "Powertrain provided force";
  SI.Force Fd "Drag force";
  SI.Force Fr "Rolling resistance";
  SI.Force Fg "Weight poryected force";
  SI.Force Fi "Inertial force";

equation
  Fd = 0.5*Cd*Af*rho*(V^2)*sign(V);
  Fr = M*g*cos(Alpha*pi/180)*Cr;
  Fg = M*g*sin(Alpha*pi/180);
  Fi = M*der(V);

  F - Fd - Fr - Fg - Fi = 0;
  F = -TorqueIN.T/R;
  der(TorqueIN.Phi) = V/(2*pi*R);


end BodyFrame1DOF;
```

EPowetrain.Mechanical.Slope

Terrain Slope

Connectors

Type	Name	Description
input InPort	H	Current Heigth
output OutPort	Alpha	Slope

input InPort	d	Current displacement

Modelica definition

```
model Slope "Terrain Slope"

  Interfaces.InPort H "Current Heigth";
  Interfaces.OutPort Alpha "Slope";

  Interfaces.InPort d "Current displacement";


equation
  when (time > 0) then
    Alpha = asin(der(H)/der(d));
  end when;
end Slope;
```

EPowetrain.Control

Contains control blocks used to implement energy management or actuator control strategies.

Package Content

Name	Description
PID	Proportional-Integra-Derivative controller

EPowetrain.Control.PID

Proportional-Integra-Derivative controller

Parameters

Type	Name	Description
Boolean	LimitOut	
Real	Max	
Real	Min	
Boolean	DeadZone	
Real	eps	Dead zone range
Real	K	
Real	I	
Real	D	
Real	Smt	Smooth factor

Connectors

Type	Name	Description
output OutPort	Out	
input InPort	In	
input InPort	Ref	

Modelica definition

```
model PID
  "Proportional-Integra-Derivative controller"

  Interfaces.OutPort Out;
  Interfaces.InPort In;
  Interfaces.InPort Ref;


  parameter Boolean LimitOut=false;
  parameter Real Max=0;
  parameter Real Min=0;

  parameter Boolean DeadZone=false;
```

```
  parameter Real eps(min=0) = 1e-9 "Dead zone range";

  parameter Real K=1;
  parameter Real I=0;
  parameter Real D=0;

  parameter Real Smt=1e6 "Smooth factor";

  Real AuxOut(start=0);
  Real Error(start=0);
  Real IntEr(start=0);

equation
  AuxOut = K*Error + D*der(Error) + I*IntEr;
  Error = Ref - In;

  // DEADZONE
  if DeadZone == true and noEvent(abs(AuxOut) < eps)
       then
    der(Out) = -Smt*Out;
    der(IntEr) = 0;
  elseif LimitOut == true then
    //ANTIWINDUP
    if noEvent(AuxOut >= Max) then
      der(Out) = Smt*(Max - Out);
      der(IntEr) = 0;
    elseif noEvent(AuxOut <= Min) then
      der(Out) = Smt*(Min - Out);
      der(IntEr) = 0;
    else
      der(Out) = Smt*(AuxOut - Out);
      der(IntEr) = Error;
    end if;
  else
    der(Out) = Smt*(AuxOut - Out);
    der(IntEr) = Error;
  end if;


end PID;
```

EPowetrain.Sensors

Contains idealized sensors for measuring physical variables (e.g., voltage, current, torque, speed), used in control feedback or performance monitoring.

Package Content

Name	Description
Vsensor	Voltage measurement sensor
AxialSpeed	Angular position and speed measurement sensor
CurrentSensor	Current measurement sensor

EPowetrain.Sensors.Vsensor

Voltage measurement sensor

Connectors

Type	Name	Description
PosPin	p	
PosPin	n	
output OutPort	outPort	

Modelica definition

```
model Vsensor "Voltage measurement sensor"

  EPowetrain.Interfaces.PosPin p;
  EPowetrain.Interfaces.PosPin n;
```

```
  Modelica.Units.SI.Voltage v;
  Interfaces.OutPort outPort;
equation
  p.i = 0;
  n.i = 0;
  v = p.v - n.v;
  outPort = v;
end Vsensor;
```

EPowetrain.Sensors.AxialSpeed

Angular position and speed measurement sensor

Parameters

Type	Name	Description
Integer	N	Pole pairs

Connectors

Type	Name	Description
MechanicalAxis	Axis_In	
MechanicalAxis	Axis_Out	
output OutPort	Wm	
output OutPort	Th_m	
output OutPort	We	
output OutPort	Th_e	

Modelica definition

```
model AxialSpeed
  "Angular position and speed measurement sensor"

  Interfaces.MechanicalAxis Axis_In;
  Interfaces.MechanicalAxis Axis_Out;
  Interfaces.OutPort Wm;
  Interfaces.OutPort Th_m;
  Interfaces.OutPort We;
  Interfaces.OutPort Th_e;
  parameter Integer N(min=1) "Pole pairs";
  constant Real pi=Constant.pi;


equation

  Th_e = N*Axis_In.Phi;
  Th_m = Axis_In.Phi;

  when (Th_e >= 2*pi) then
    reinit(Th_e, 0);
  end when;


  Wm = der(Th_e);
  We = der(Th_m);
  connect(Axis_In, Axis_Out);

end AxialSpeed;
```

EPowetrain.Sensors.CurrentSensor

Current measurement sensor

Connectors

Type	Name	Description
PosPin	P	

NegPin	N	
output OutPort	Imeas	

Modelica definition

```
model CurrentSensor "Current measurement sensor"

  Interfaces.PosPin P;
  Interfaces.NegPin N;
  Interfaces.OutPort Imeas;


equation
  P.v = N.v;
  N.i = Imeas;
  P.i + N.i = 0;


end CurrentSensor;
```

EPowetrain.Examples

Provides usage examples and test benches for validating component behavior under specific driving conditions or scenarios.

Package Content

Name	Description
UDDS_Cycle	Example experiment under UDDS drive cycle
Trip	Example experiment
DCMotor	Pasek's vs Moments motor calibration validation

EPowetrain.Examples.UDDS_Cycle

Example experiment under UDDS drive cycle

Modelica definition

```
model UDDS_Cycle
  "Example experiment under UDDS drive cycle"
  Electrical.Devices.Converters.ElectricConverter
    electricConverter;
  Mechanical.BodyFrame1DOF bodyFrame1DOF(
    M=1500,
    Af=2.2,
    Cd=0.29,
    Cr=0.009,
    rho=1.2);
  Electrical.Sources.Ground ground2;
  Electrical.Sources.Ground ground1;
  Control.PID pID(
    LimitOut=false,
    Max=1,
    Min=-1,
    K=20,
    I=0.1);
  Sources.UDDS uDDS;
  Modelica.Blocks.Math.Gain mph_to_ms(k=0.44704);
  Sources.Constant Constant(Value=0);
  Electrical.Sources.Battery battery(
    Cap=30,
    Vd=300,
    Vf=420,
    InitSOC=0.8);
  Electrical.Devices.Machines.DCMotor dCMotor_2_1(
    Rm=0.025,
    Lm=1e-3,
    Ke=1.4,
    Kt=1.4);
equation
  connect(ground2.p, electricConverter.N_Out);
  connect(pID.Out, electricConverter.DutyCycle);
```

```
  connect(uDDS.y, mph_to_ms.u);
  connect(mph_to_ms.y, pID.Ref);
  connect(bodyFrame1DOF.V, pID.In);
  connect(Constant.Out, bodyFrame1DOF.Alpha);
  connect(battery.posPin, electricConverter.P_In);
  connect(battery.negPin, ground1.p);
  connect(battery.negPin, electricConverter.N_In);
  connect(electricConverter.P_Out, dCMotor_2_1.Vp);
  connect(electricConverter.N_Out, dCMotor_2_1.Vn);
  connect(dCMotor_2_1.Rotor, bodyFrame1DOF.TorqueIN);
end UDDS_Cycle;
```

EPowetrain.Examples.Trip

Example experiment

Modelica definition

```
model Trip "Example experiment"
  Electrical.Devices.Converters.ElectricConverter
    electricConverter;
  Mechanical.BodyFrame1DOF bodyFrame1DOF(
    R=0.29,
    M=1280,
    Af=2.38,
    Cd=0.29,
    Cr=0.0084,
    rho=1.225);
  Electrical.Sources.Ground ground2;
  Electrical.Sources.Ground ground1;
  Control.PID Driver(
    LimitOut=true,
    Max=1,
    Min=-0.5,
    K=0.5,
    I=0.005,
    D=0.1,
    Out(start=1));
  Modelica.Blocks.Sources.CombiTimeTable Cycle(table=
        fill(0.0, 0, 2));
  Electrical.Sources.Battery battery(
    Cap=60,
    Vd=335,
    Vf=360,
    InitSOC=0.869,
    Rs=0.06,
    Imax=400);
  Modelica.Blocks.Math.Gain kph2ms(k=1/3.6);
  SignalRouting.Terminator DataSOC;
  Mechanical.Slope slope;
  Modelica.Blocks.Continuous.Integrator integrator;
  SignalRouting.Terminator DataIbatt;
  SignalRouting.Terminator DataTorque;
  Mechanical.Basic.Gearbox gearbox1(N=1/6);
  Electrical.Basic.Resistance RLoad(R=80.4765);
  Electrical.Devices.Machines.DCMotor dCMotor(
    Rm=1.72,
    Lm=106.26e-6,
    Ke=0.7144,
    Kt=0.72,
    bm=5e-4,
    J=31e-3);
equation
  connect(dCMotor.Vp, electricConverter.P_Out);
  connect(ground2.p, electricConverter.N_Out);
  connect(Driver.Out, electricConverter.DutyCycle);
  connect(dCMotor.Vn, electricConverter.N_Out);
  connect(bodyFrame1DOF.V, Driver.In);
  connect(battery.posPin, electricConverter.P_In);
  connect(battery.negPin, ground1.p);
  connect(battery.negPin, electricConverter.N_In);
  connect(kph2ms.y, Driver.Ref);
  connect(Cycle.y[1], kph2ms.u);
  connect(DataSOC.inPort, Cycle.y[7]);
  connect(slope.Alpha, bodyFrame1DOF.Alpha);
  connect(Cycle.y[2], slope.H);
  connect(integrator.y, slope.d);
  connect(integrator.u, bodyFrame1DOF.V);
```

```
  connect(DataIbatt.inPort, Cycle.y[6]);
  connect(DataTorque.inPort, Cycle.y[4]);
  connect(dCMotor.Rotor, gearbox1.AxisA);
  connect(gearbox1.AxisB, bodyFrame1DOF.TorqueIN);
  connect(RLoad.n, electricConverter.P_In);
  connect(RLoad.p, electricConverter.N_In);
end Trip;
```

EPowetrain.Examples.DCMotor

Pasek's vs Moments motor calibration validation

Modelica definition

```
model DCMotor
  "Pasek's vs Moments motor calibration validation"
  Electrical.Devices.Machines.DCMotor Moments(
    Rm=30.9,
    Lm=0.803,
    Ke=1.323,
    Kt=1.323,
    bm=0.0005,
    J=0.0031);
  Electrical.Sources.VStep vStep(
    St=0.5,
    v0=60,
    vf=248);
  Electrical.Sources.Ground ground;
  Electrical.Devices.Machines.DCMotor Pasek(
    Rm=30.9,
    Lm=0.438,
    Ke=1.323,
    Kt=1.323,
    bm=0.0005,
    J=0.0036);
equation
  connect(vStep.n, Moments.Vn);
  connect(vStep.p, Moments.Vp);
  connect(ground.p, Moments.Vn);
  connect(Pasek.Vp, vStep.p);
  connect(Pasek.Vn, vStep.n);
end DCMotor;
```

Automatically generated Wed May 28 17:57:10 2025.