Politecnico di Torino

Aerospace Engineering

A.a. 2025/2026

Graduation Session July 2026

Automatic Virtual Verification of Flight Control System Requirements

Supervisors:

Politecnico di Torino

Roberta Fusaro

Davide Ferretto

German Aerospace Center

Francesco Torrigiani

Carlos Cabaleiro De La Hoz

Candidate

Davide Magris

Abstract

The increasing complexity of aeronautical systems, combined with the growing need for a more sustainable aviation, requires more efficient verification and validation workflows. In the current state of the art, the V-model development processes rely on Document-Based System Engineering and manually generated simulation models that are loosely coupled with the system architecture, leading to inconsistencies, poor traceability, and significant rework and higher costs when design changes occur.

This thesis proposes a novel Model-Based System Engineering workflow for the automated virtual verification of complex aeronautical systems, developed in the context of the ODE4HERA project in the Clean Aviation initiative for the development of a Hybrid-Electric Regional aircraft. The scope is to automate the generation and execution of dynamic simulations from system architecture models enabling a "Left Shift" of the verification and validation activities in the early design phases, before physical prototypes are available.

The proposed methodology uses SysML v2 to formalize both the system requirements and the derived architecture. A Model-Based Design architecture is derived by importing from Modelica component libraries into SysML v2. This architecture is then enriched with simulation specific components to create a test-bench for the verification of a specific requirement. Starting from this, a custom Python tool parses the SysML v2 model through a JSON intermediate representation and automatically generates simulation-ready Modelica models using the modelica-builder library. The simulations are then executed via OMPython, results are extracted and the requirement verification status is automatically updated in SysML v2 without manual intervention.

The methodology is validated through an application case on the Flight Control System of a conventional aircraft. Dynamic models are developed for Electro-Mechanical Actuators (EMA) and Electro-Hydrostatic Actuators (EHA), sized on the Airbus A320 specifications and integrated in the model of the elevator control surface. Static requirements as maximum no-load velocity, maximum hinge moment and steady-state-error, as well as dynamic requirements such as the bandwidth of the system can be verified.

The results show that the proposed methodology allows the automatic generation of dynamic models starting from system requirements and architecture, without manual intervention in the simulation environment. An important aspect is the decoupling between the Model-Based design architecture, which is defined once for all requirements, and the simulation architecture, which are generated separately

for each requirement. The verification status is automatically updated in SysML v2 using the simulation results, closing the loop of the design process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

For many years, the discipline of system engineering had to face the increasing in complexity inherent in the development of complex system, as it can be a product in the aerospace sector. To ease the work a standard operational framework was developed to conceptualize, design and develop these complex systems: the V-model diagram 1.1. The V-model is simply a method to ease the system complexity by structuring the development process into smaller and more manageable tasks.

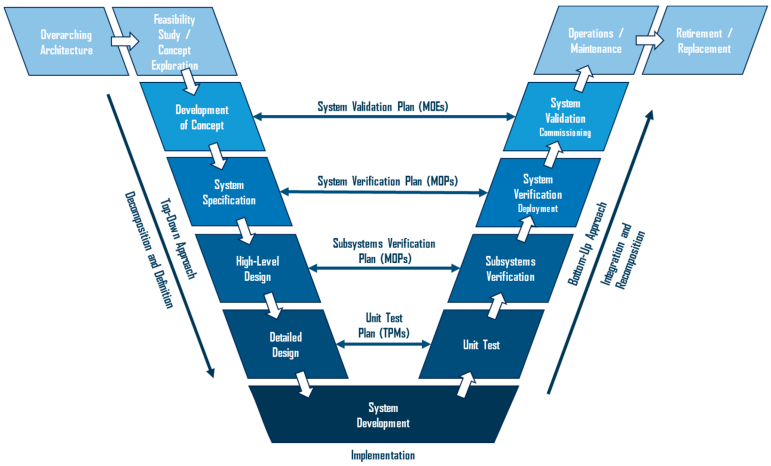

Figure 1.1: V model implemented in a typical development process

As can be seen in the diagram, the left descending branch of it dictates a top-down flow that translate high level stakeholder needs into lower and more formal system requirements that defines the high level system architecture and subsequently in more detailed elements. The bottom of the V-model is the actual

development and production of the different components into which the system has been decomposed. The right ascending branch govern the bottom-up integration and testing phases. Each hierarchical level is here verified and validated ($V\&V$) in order to ensure that the physical system strictly aligns to the conceptual design.

It can be immediately noted that this approach has some risks, especially economical ones. The test and validation phase is carried out on the right branch of the model, so only after the system, or at least its components, have been built. If a discrepancy is found here and the actual system perform worse than expected there are only two possibilities: one we change the component/system or we lower the requirements. Changing a component/system later in the design process is very expensive because it needs a re-design, a new development (with all the cost due to manufacture and integrate) and tests. In the other hand, lowering the requirements is no much better and can be even worse. It signify that the company has failed in satisfy the stakeholder needs that can refuse to buy the product if the damage may compromise his revenue. In the defense sector it can ultimately result in the termination of the procurement contracts.

We are all familiar with the diagram in fig 1.2: the life cycle costs tend to be established early in the design process. The curve shows that the late identification and related fixes to problems cost considerably more later in the life cycle. Studies show that while the design phase absorbs only approximately 15% of total expenditure, the decisions made during this period lock in roughly 75% of the overall life-cycle costs.. With this in mind, it makes perfect sense for a company to invest more money in the design process, finding errors in these early phases is a key enabler to save money and time.

In order to improve the limitation of the traditional V-model a new approach has been developed and is shown in fig 1.3. The core innovation consist in anticipating the $V\&V$ activities traditionally relegated to the right branch. To enable this "Left Shift" strategy a systematic integration with dynamic simulation models in early design phases is a key enabler. by adopting a Model-in-the-loop (MiL) approach the system behavior is validated against the requirement in a virtual environment before physical prototypes are available.

In the current state-of-the-art though, the generation of dynamic simulation models still requires manual effort and is often loosely coupled with the system architecture definition. This is mainly due to how information is stored and transferred throughout the development process. The decomposition of the system design and development into a V-model requires, as much as any other approach, a very precise and complete documentation of every aspect of the process. Everything starting from the stakeholders needs to the requirements, architectures and so on must be documented. Today this is carried out with something to which we can address as Document Based System Engineering (DBSE). As can be foreseen there are several issues related with this methodology:

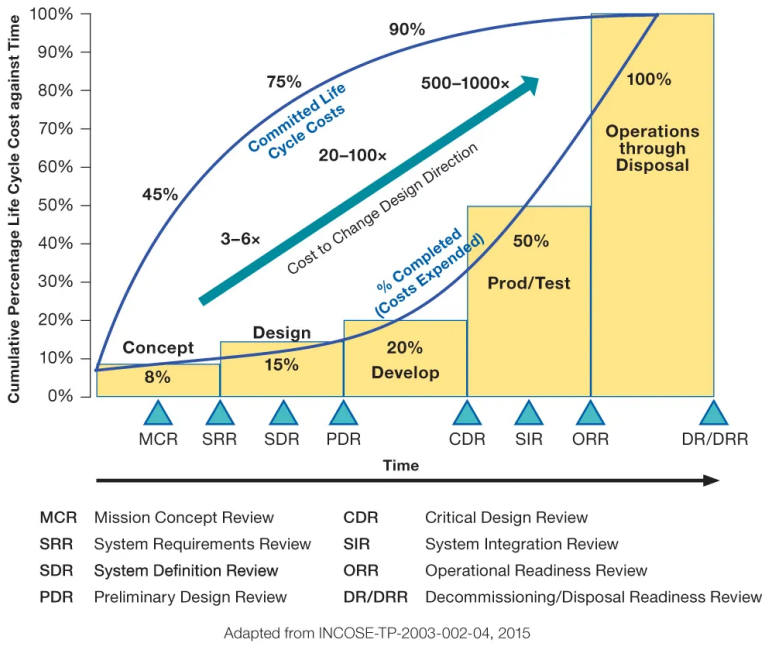

Figure 1.2: Life-cycle cost impacts from early phase decision-making [1]

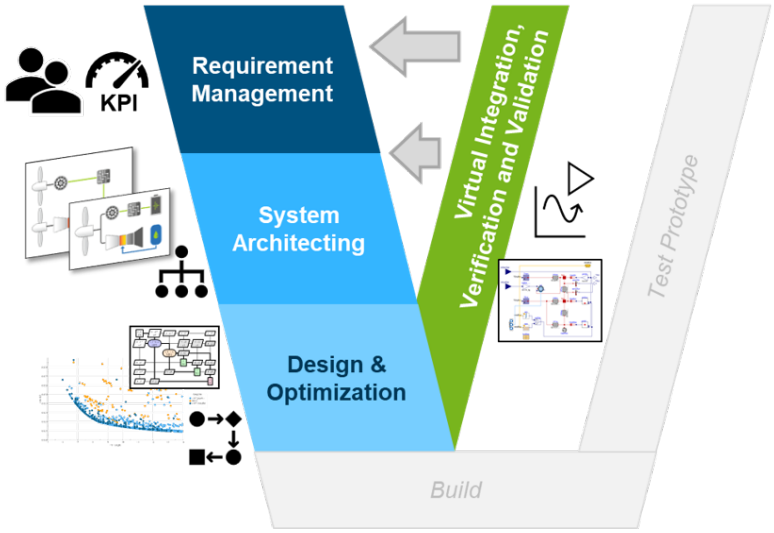

Figure 1.3: Proposed V-model with early $V\&V$ [2]

- **Project workflow and planning:** A huge effort must be put in order to have consistent documents inside a project: writing, reviewing, updating are all steps that must be carried out. Also, once a document is published it must then be frozen in order to be released and to maintain consistency with subsequent work. Sometimes, the document may even become obsolete after publication due to subsequent developments and the updates are not available until the next release.
- **Traceability:** It can be hard to trace-back information across documents, especially considering the different release of the document over time. This may cause inconsistencies in the design process and later on errors that have to be fixed
- **Communication:** Communication between teams or with the stakeholders is supported by documents and additional drawings, presentations and so on. Since the information is so spread out it can be difficult to understand.

To overcome almost all these problems at once, a new approach to design has been developing in the past years: Model Based System Engineering (MBSE).

> *"Model-based systems engineering (MBSE) is the formalized application of modeling to support systems requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases."* [3]

The key idea behind MBSE is the description of the system as a model that automatically analyses and validate the system itself. MBSE provides also a link between various domain specific tools that all converge together to produce a model-based framework for a system engineering project. MBSE can be discussed in terms of its 3 fundamental pillars: modeling language, tools and methodology. The tool is simply the software used to produce the model that consists in elements, diagrams, tables etc. The Object Management Group Modeling Language (SysML) has emerged as a state-of-the-art for system engineering since it is well suited to capture MBSE activities.

1.2 Scenario

The aerospace field is a highly energy-demanding domain, contributing to roughly 2% of the global CO2 emissions in 2022. If no countermeasures are taken, the International Civil Aviation Organization (ICAO) estimations show that by 2050 the emissions from international flights might reach three times the values recorded in 2015. Aviation also accounts for 13.9% of transport emission, and is second only to road transport. As can be seen by looking at figure 1.4 the current trend for the aviation industry is the continuous increase in CO_2 and other greenhouse gas emissions. To combat the Green House effects due to CO_2 emissions all sectors, including aviation, need to contribute and improve or find new solutions in order to achieve the EU's climate target. In 2019 the EU launched the European Green Deal.

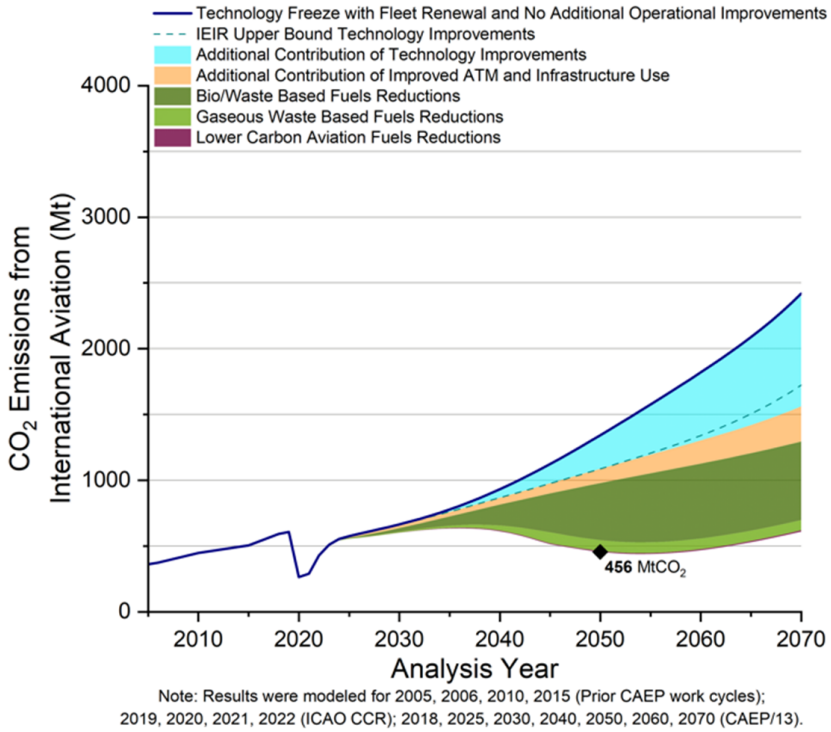

Figure 1.4: Combustion CO_2 emission, 2005 to 2070, ICAO [4]

To translate the directives of the European Green Deal into an operational and industrial framework, the European Union established the Clean Aviation Joint Undertaking (CAJU). This public-private partnership represents the primary program for the transition towards sustainable and climate-neutral aviation.

Funded as part of Clean Aviation there is the project for an Open Digital

Environment for Hybrid-Electric Regional Architectures (ODE4HERA).

The core objective of the ODE4HERA project is to drive and speed up the engineering process of hybrid-electric regional (HER) aircrafts, aiming for their entry to service in 2035. These novel configurations involve breakthrough technologies that bring a systemic complexity far superior to conventional aeronautical architectures. Since the current digital frameworks are not capable of managing such demands, ODE4HERA's main task is the creation of a transferable Open Digital Platform (ODP), figure 1.5. This platform integrates heterogeneous software tools across the entire aircraft development cycle by processing data through MBSE. The ODE4HERA ODP enables automated frontload verification during the design conception phase and virtualized validation. This operational capability is a critical engineering requirement to streamline future certification processes for hybrid-electric aeronautical architectures.

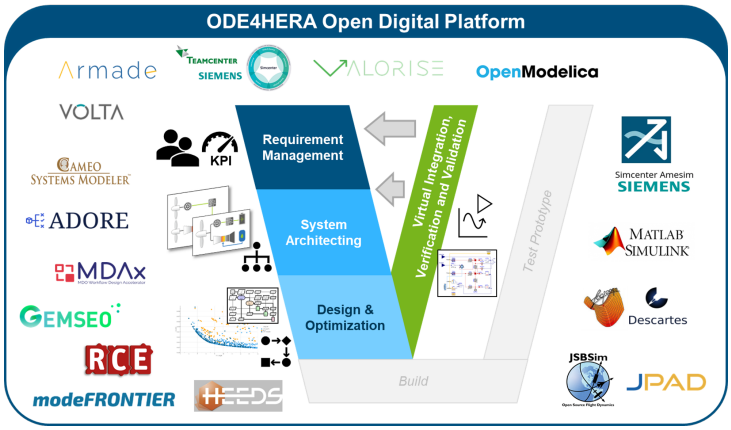

Figure 1.5: ODE4HERA Open Digital Platform

The work of this thesis is therefore part of the ODE4HERA project and will investigate the potential of OpenModelica and SysMLv2 for the process of automatic $V\&V$ of requirements.

1.3 Research Question

Having defined the motivation behind this work and the scenario within it is placed, it is possible now to formulate the research objective of the present thesis. Before stating the Research Question it is to be noted that not everything that will be presented is completely new (see chapter 2). The Dassault Systemes product CATIA has a proprietary plugin for SysML support that also enable direct import/export SysML files to Modelica models. Not only this, in the work of *Pepper, B. and Arifin, H. H. and Pavalkis, S. and Matam, J. and Kratzke, R.* [5] a bi-directional tranformation between SysML and Modelica for dynamic simulation in an $V\&V$ framework was investigate and a methodology proposed. It can be observed though, that there is a lack of studies and tools for SysMLv2 and that almost all the state of the art only focuse on the old version of SysML. In addition, a valid methodology for leveraging the translation from SysML to Modelica in an automatic $V\&V$ framework is not yet been developed. Keeping in mind this the research question can be formulated as:

"How can we streamline and automate the generation and execution of dynamic simulations directly from system architecture models in order to verify system requirements?"

Chapter 2

State of the Art and Background

2.1 SysML Multi-Domain Simulation Integration

In the current state of the art many extensions for bridging sysML with Modelica exist like SysML4Simulink, SysML4Arena and SysML4Modelica. It is very important to note that an extension to bridge SysML and Modelica already exist but it is not yet supporting the new version of SysMLv2. All these extensions contain additional semantic and stereotypes to allow the creation of simulation language-specific models in SysML. In *Multi-domain simulation utilizing SysML: state of the art and future perspectives [6]* an in depth analysis of SysML Simulation integration is performed and some of the key aspects are here reported. In fig 2.1 it is possible to see a simplified approach for a general integration. As can be seen, the sysML model is enriched using a meta-model, simulation specific extension profiles and model libraries.

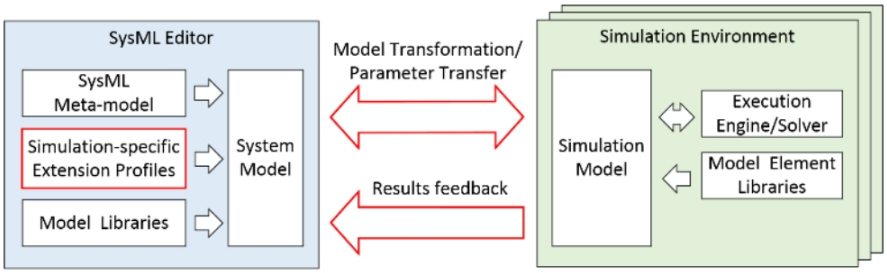

Figure 2.1: SysML-Simulation Setup [6]

Once the SysML model and the required simulation libraries are set up, it

becomes necessary to feed the simulation environment with all the data needed to run the model. This includes boundary conditions, initial states, design parameters and many equations or constraints that govern the system: all of this information can be stored directly in SysML and retrieved at runtime. Regarding how this data exchange is performed, two main strategies are documented in [6]: the Model-to-Text (M2T) and the Model-to-Model (M2M) approaches, illustrated in figure 2.2. The M2M route is generally preferred because it relies on formal meta-model mappings between the two environments, resulting in a more robust and standardized bridge. In practice, M2M is almost always followed by a downstream M2T step to produce the final executable code; however, this last conversion is already handled natively by tools such as OpenModelica or OMPython, so it does not require additional effort from the engineer.

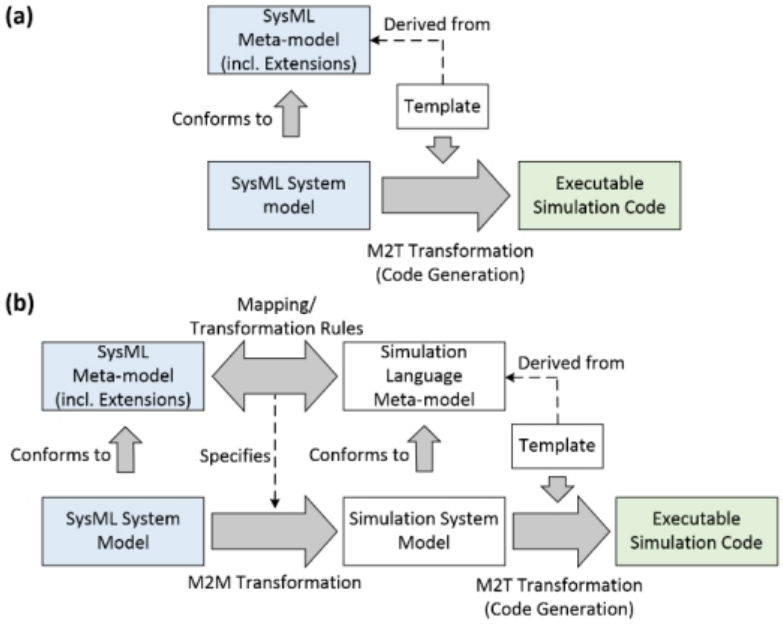

Figure 2.2: a) M2T vs b) M2M transformation approach [6]

2.1.1 SysML to Modelica transformation specification - CATIA SysML plugin

To close the gap between architecture design and dynamic modeling, the Object Management Group formalized a SysML-Modelica Tranformation Specification, a standard to define a formal bi-directional semantic mapping between the two languages. Since both the languages follows the object-oriented paradigm, the

resulting structure is similar and thus, it is possible to derive an ontological equivalence. The practical transformation is achieved by adding specific stereotypes (ex: «modelicaModel», see figure 2.3) to standard SysML elements. Having standardized the mapping it is possible to utilize existing Modelica libraries directly in SysML.

Table 2.1: SysML4Modelica semantic mapping - main elements

SysML Construct	Modelica Target	SysML4Modelica Stereotype
Block	Class, Model, Block	«modelicaClass» / «modelicaModel»
Port (Proxy Port)	Connector (Acausal)	«modelicaConnector»
Connector (IBD)	Connection, Equation	«modelicaConnection»
Constraint Block	Equation	«modelicaEquation»
Value Property	Variable, Parameter	«modelicaVariable» / «modelicaParameter»

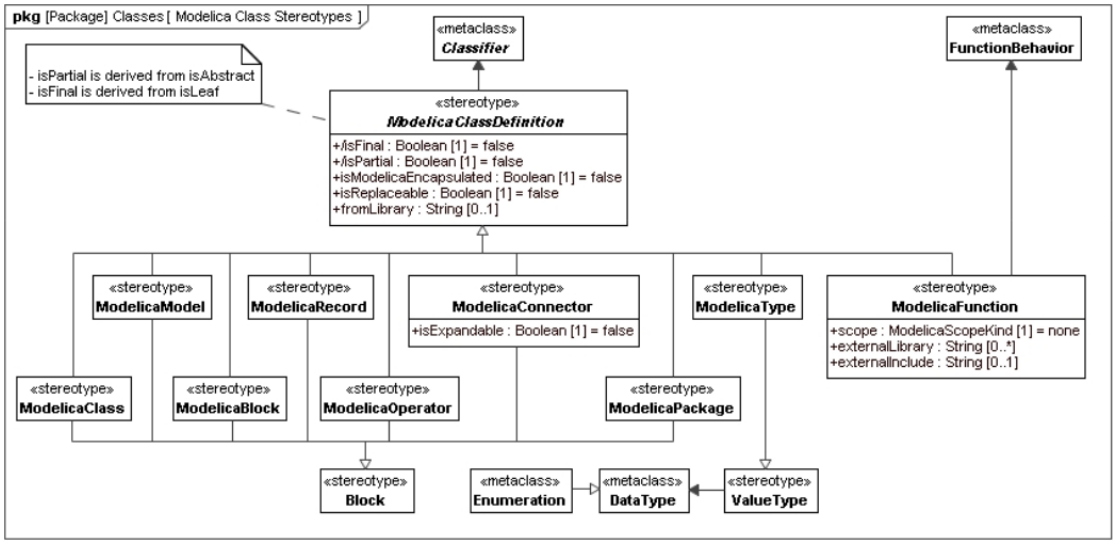

Figure 2.3: SysML4Modelica Stereotypes

A very relevant example of how the specification is used is the CATIA SysML plugin. This plugin enables the user to import from and export to External Simulation Tools as Simulink or Modelica. Specifically for the Modelica export, any existing model in sysml can be transformed into Modelica but, if exporting also the behavior, additional annotation are required. SysML properties must be marked

with the OMG SysPhs specification stereotypes, for example if a SysML property is annotated with «PhSConstant» it will be exported as a Modelica parameter. For a better understanding, we refer to the SysML 2024x plugin documentation [7], but some of the main feature are reported here because useful for the purpose of this thesis:

- **Partial Model export:** Export parts, ports and connection defines inside an IBD to a Modelica file (.mo).
- **Referencing existing Modelica componets:** possible by adding a specific tag to the component with the Modelica reference path, see figure
- **Imports of Modelica libraries in SysML**
- **Simulation of exported models:** possible by defining simulation setup in CATIA
- **Extending models with comments:** useful to add properties or specialization to a component.

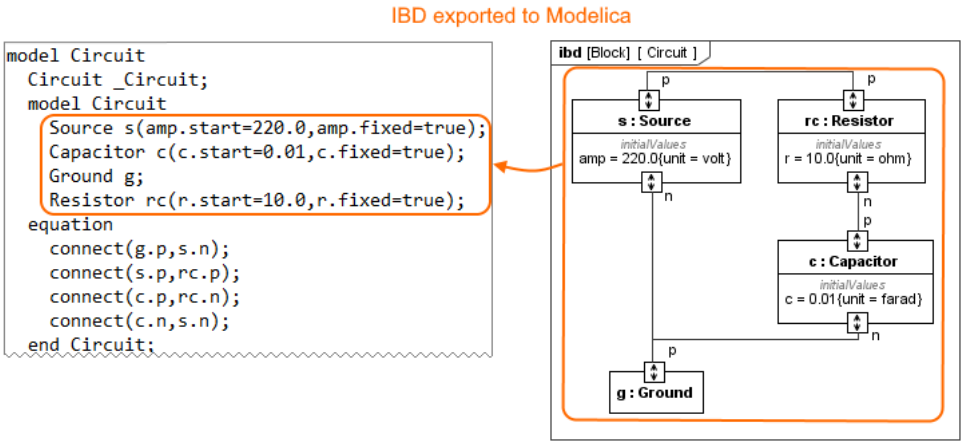

Figure 2.4: Internal Block Diagram export to Modelica, CATIA [7]

2.1.2 SysML to Modelica for requirements verification

The integration of SysML with Modelica is nowadays achieved mainly with two approaches (Brian Pepper, Habibi Husain Arifin, Saulius Pavalkis, Jyothi Matam, Ronald Kratzke, INCOSE international symposium [5]):

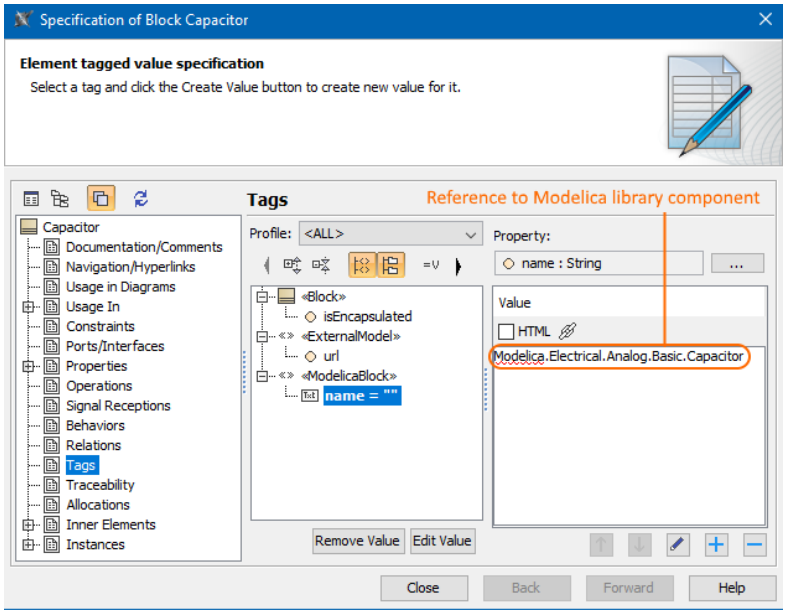

Figure 2.5: Referencing Modelica library component, CATIA [7]

- **SysML-Modelica Bi-Directional Transofrmation:** This approach is based on a transformation specification between the two languages. The most prominent extension is SysML4Modelica profile. Since SysML itself lacks the capability to model physical interaction between systems another extension as SysPhS (SysML Extension for Physical Interaction and Signal Flow Simulation) is necessary to extend SysML with additional information.
- **Co-Simulation between SysML and Modelica:** This approach allows the concurrent execution of models in their respective environments by compiling either SysML or Modelica models as .fmu through FMIs.

The integration of SysML and Modelica into a MBSE approach in order to perform early stages trade-studies analysis and $V\&V$ requires a systematic workflow to transform the architecture models in a proper way. A workflow description is provided always in (Brian Pepper, Habibi Husain Arifin, Saulius Pavalkis, Jyothi Matam, Ronald Kratzke, INCOSE international symposium [5]) and is here proposed (figure 2.6) since a revised approach has been used in the methodology of this work:

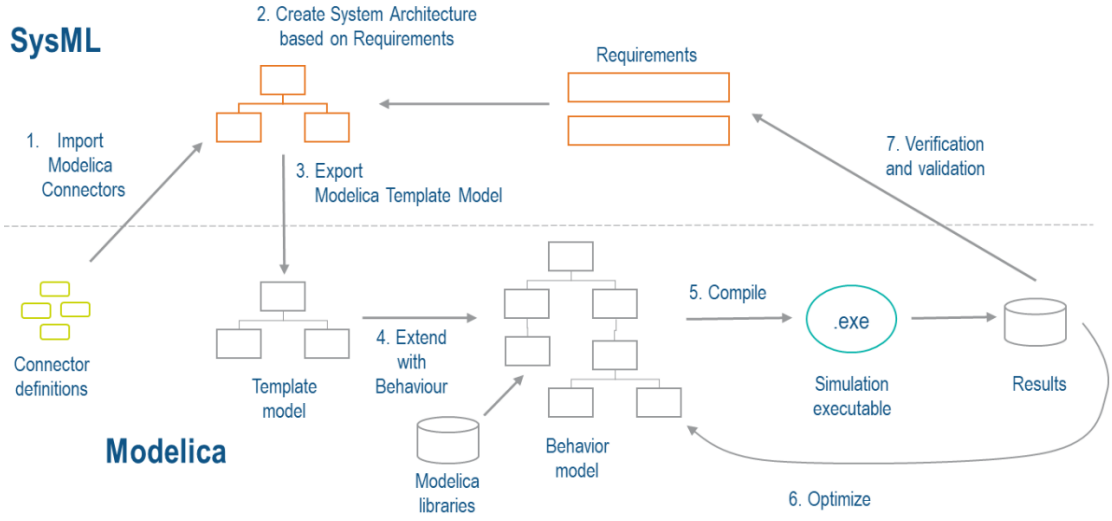

Figure 2.6: SysML to Modelica Integration Worflow [5]

- **Import Modelica connectors:** Physical connectors and interfaces from Modelica are imported in SysML as well as components from the Standard Library
- **Creation of System Architecture from Requirements:** After requirements are defined the system architecture can be modeled with different levels

of detail starting from a conceptual view to a more detailed physical representation. The physical architecture enriched with the imported components provides the first template model.

- **Export Modelica Template Model:** SysML artifacts are converted in Modelica blocks, ports, parameters connectors etc. utilizing the SysPhS extension.
- **Extend with behavior:** The physical behavior of the system is now added to the generated template components.
- **Simulate and Optimize:** At this point the models extended with the behavior are compiled using Modelica symbolic manipulation and simulated. Engineers can now iterate multiple simulations to optimize parameters and behavior to be sufficiently realistic.
- **Verification and Validation:** The outcome of the simulations in Modelica can be fed-back to SysML to validate and verify requirements completing the process.

2.2 Virtual verification in the software industry

In the early 2000's the advent of the Unified Modeling Language (UML) brought a shift of paradigm in the software development life-cycle (SDLC). Model Based Design was introduced as a new approach that uses models directly as primary artifacts throughout the SDLC rather than using source code [8]. In 2001 the Object Oriented Group (OMG) proposed this new paradigm called Model Driven Architecture (MDA) that uses subsets of the UML as state machines and class diagrams and automatic code generation from these models in order to decrease the development effort. Automatic Code Generation became a standard in many sectors, especially in aerospace, because it has many benefits as being less error prone, easier to maintain, more accurate and can be generated by people specialized in other fields. This is very attracting for all the safety-critical applications where developer are usually domain experts with less background in programming fault-tolerant real time systems, one of these fields is, for example, controllers.

2.2.1 Automatic code generation

The core principle involves constructing system models using standardized notations, such the UML, prior to physical implementation. The model is than compiled and executed to validate the system behaviour pre-deployment. The system modeled using UML subsets can than be automatically translated to executable code via tools called "code generators", The most useful subset is the State Chart Diagram since it can generate code independently. In the work of Sunitha E.V. and Philip Samuel [9] the automatic code generation for State Chart Diagrams was investigated and an example is reported in fig 2.7 as reference.

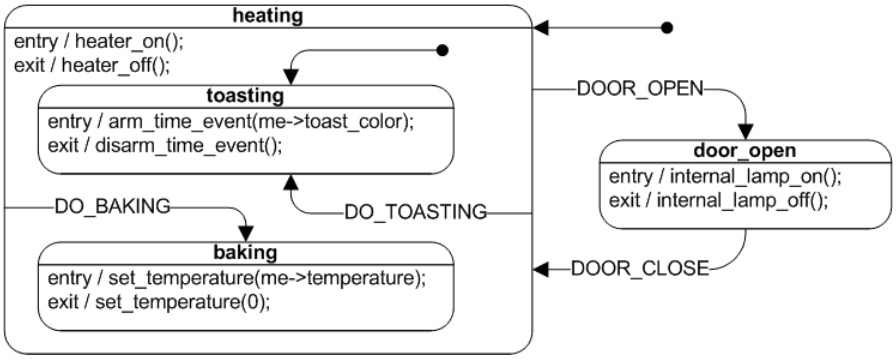

Figure 2.7: UML state diagram representing a simple oven

The automatic code generation starting from the graphical models is a structured frameworks based on abstraction levels and formal transformations. Following MDA guidelines, the code generation follows three intermediate steps:

- **Platform Independent Model (PIM):** This is the starting representation of the system built in UML, fully abstract and decoupled from any implementation technology. A State Chart Diagram is a typical example of a PIM artifact.
- **Model-to-Model Transformation (M2M):** A dedicated transformation engine takes the PIM as input and produces a Platform Specific Model (PSM) by injecting platform-dependent rules, constraints and configuration tags. The PSM is no longer abstract: it is already tied to a specific target technology, making it a Technology-Specific Design.
- **Model-to-Text Transformation (M2T):** The PSM is passed to a second engine that maps each model element to a textual template, ultimately producing the source code in the desired language (Java, C/C++, etc.).

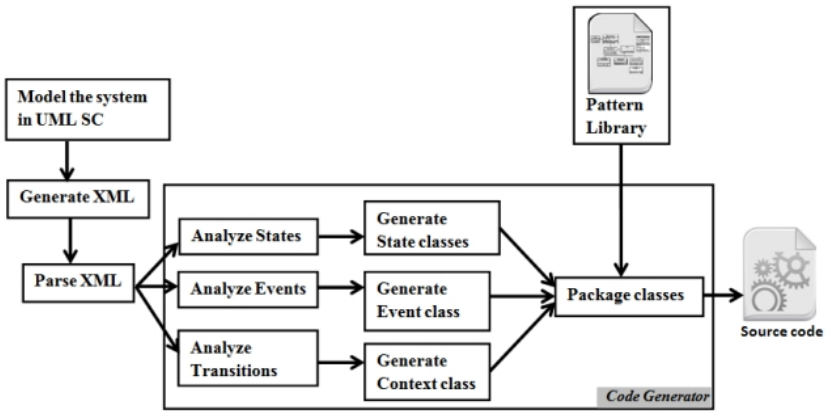

Figure 2.8: Architecture of Code Generator [9]

Another approach for automatic code generation involves the use of models. A number of tools for model-based development is nowadays available like Matlab/Simulink by Mathworks or SCADE. In Matlab/Simulink, C/C++ code can be automatically generated out of models. In these tools the generation ability it's outsourced to templates, therefore the code generator task is limited to the selection of the appropriate template 2.9. A templates is simply an application independent solution for one aspect of the system.

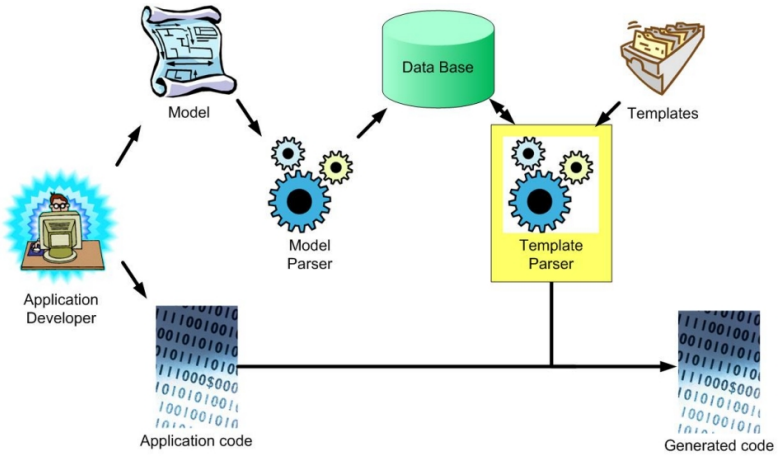

Figure 2.9: Model-Based code generation process [10]

2.3 Modelica Language

Modelica is a modeling, object-oriented, language based on equation and developed for the simulation of complex dynamic systems. Modelica is not a procedural programming language, it describes the mathematical relations that describes the overall system behaviour and is, for this reason, acasual. This architecture allows the possibility of modeling multi-domain systems like CPS (Cyber-Physical Systems) where different physical domains interacts in a homogeneous way.

2.3.1 Acausual Modeling

To understand the concept of acausal modeling, an example is well suited. Assume that we want to study the motion of a system composed of a sliding body with a pendulum attached to it, as shown, for example, in fig. 2.10. As a first step, the system will be solved to find the law of motion "the classic way," and then it will be shown how the same result can be achieved with Modelica.

Figure 2.10: acrobat hanging on a pendulum mounted to a motorbike

We can assume that only the vertical dimension and the one along the wire are relevant for this study; in this way, the motion has been reduced to two dimensions. we can also assume that the wire can be regarded as rigid and that the bodies can be represented as point masses. With these simplifications, we can then study the forces acting on both the motorcycle and the hanging acrobat, as shown in fig. 2.11.

Skipping some basic derivations, the following system of 12 equations for 12 unknowns can be obtained:

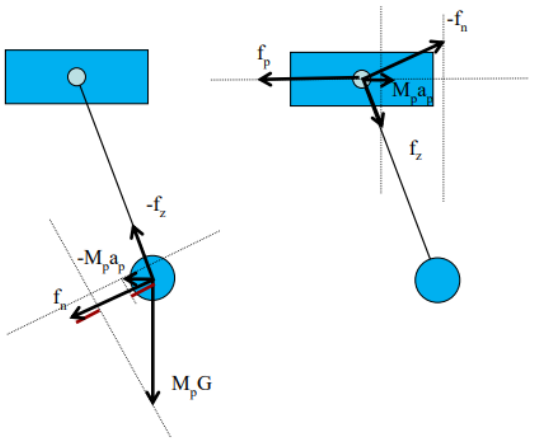

Figure 2.11: two dimensional simplification and acting forces

$$\begin{cases} a_S = \dot{v} \\ v = \dot{s} \\ f_S = M_S a_S \\ \alpha = \dot{\omega} \\ \omega = \dot{\varphi} \\ \tau = I\alpha \\ \tau = f_n R \\ f_n = \sin(\varphi) M_P G - \cos(\varphi) M_P a_P \\ f_P + \sin(\varphi) f_z - \cos(\varphi) f_n - M_P a_P = 0 \\ f_z = M_P R \omega^2 \\ a_S = a_P \\ f_S + f_P = 0 \end{cases} \tag{2.1}$$

To solve the system of equations, it is necessary to causalize it and bring it to state-space form. After a few mathematical steps that are not relevant for the purpose of this thesis, the algebraic solution of the system can finally be found, where a is the acceleration of the motorcycle:

$$a = \frac{\sin(\varphi) f_z - \cos(\varphi)\sin(\varphi) M_P G}{M_S + M_P(1 - \cos(\varphi)^2)} \tag{2.2}$$

At this point, if we want to see what the motion of the system looks like, we need to integrate the solution over time using a method of our choice, such as Forward Euler or a Runge-Kutta method.

The process was rather laborious even for such a simple example, and if the model is changed, the process has to be redone entirely. For these reasons, and many more, a number of programming languages have been developed with the aim of automating this process; one of these is Modelica.

Modelica can handle non-causal equations such as $R \cdot I = V$, which means that the order of computation is not decided at modeling time, allowing the model equations to be written directly with no need to derive the state-space form by hand.

Table 2.2: Comparison between Acausal and Causal modeling

	Acausal	**Causal Possibilities**

Equation Level	`R*i = v;`	`i := v/R;`
		`v := R*i;`
		`R := v/i;`

Considering these elements, the resulting Modelica model can be written as follows:

```
model CartPendulum
  parameter Real MS = 1.0 "Cart mass [kg]";
  parameter Real MP = 0.5 "Pendulum mass [kg]";
  parameter Real R  = 1.0 "Arm length [m]";
  parameter Real G  = 9.81 "Gravity acceleration [m/s^2]";
  parameter Real I  = 0.5 "Moment of inertia [kg*m^2]";
  // Cart (Slider)
  Real s(start=0, fixed=true) "Cart position [m]";
  Real v(start=0, fixed=true) "Cart velocity [m/s]";
  Real aS "Cart acceleration [m/s^2]";
  Real fS "Force on cart [N]";
  // Pendulum
  Real phi(start=0.1, fixed=true) "Angle [rad]";
  Real omega(start=0, fixed=true) "Angular velocity [rad/s]";
  Real alpha "Angular acceleration [rad/s^2]";
  Real tau "Torque [N*m]";
  // forces
  Real fn "Normal/tangential force [N]";
  Real fP "Pendulum horizontal reaction force [N]";
  Real fz "Centrifugal force [N]";
  Real aP "acceleration [m/s^2]";
equation
  aS = der(v);
  v  = der(s);
  alpha = der(omega);
  omega = der(phi);
  fS = MS * aS;
  tau = I * alpha;
  tau = fn * R;
  fn = sin(phi) * MP * G - cos(phi) * MP * aP;
  fP + sin(phi) * fz - cos(phi) * fn - MP * aP = 0;
  fz = MP * R * omega^2;
  aS = aP;
  fS + fP = 0;
end CartPendulum;
```

Listing 2.1: Modelica implementation of the Cart-Pendulum system.

Once the model is ready, Modelica has to perform a series of operations in order to simulate it that are usually carried out by an environment as can be OpenModelica, Dymola or Wolfram SystemModeler, the workflow is described in fig 2.12.

1. **Modelica Model** Source Code generated by the user as shown in list 2.1 (file .mo). It's object oriented and acasual, at this state is written as system of connected objects or non directly solvable equations.

2. **Elaboration/Flattening:** The environment compiler flattens the system of equations removing objects and substituting connection between objects with coupling equations (Kirchoff law).

3. **Hybrid DAE:** Mathematical representation of flattened system. It is an implicit, non ordered system in the form of $F(\dot{x}, x, y, u, t) = 0$, where x are the states and y the algebraic variables.

4. **Equation Transformation and Code Generation:** Operates a Topological Analysis to determine which variables are obtained from which equations. Orders the equations in a solvable causal sequence (usually a lower triangular matrix). Index Reduction with Pantelides algorithm to derive analytically the equations and reduce the differential order to 1 or 0. Breaks algebraic loops and then generates the executable C/C++ code.

5. **Executable:** It is a binary file (.exe on Windows) generated from the C/C++ source code and contain the function $f(x, t)$ and the integration algorithm as DASSL, CVODE, Runge-Kutta etc...

6. **Simulation and Simulation Result:** In this phase the simulation is runned and it generate an output file (.mat or .csv) with the trajectory of all the variable in time.

The main advantage is that the engineer can focus on describing the law of the physical model without having to write or reduce the system of equations.

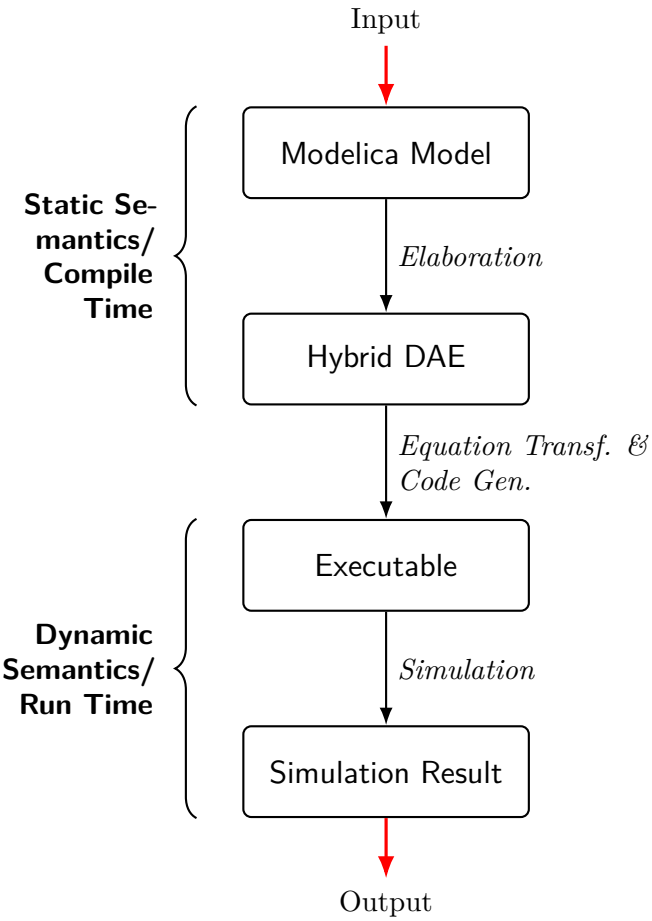

Figure 2.12: Modelica Compilation logic

2.4 SysMLv2

In the last years, as introduced in chapter 1, the Model Based System Engineering slowly became the standard methodology, especially in development of aerospace systems, were complexity is very high. For many years, the standard for the modeling language has been the first generation of SysML. Since this language was based on the Unifed Modeling Language, developed mainly for software engineering, it has demonstrated mant inner limitation when it had to describe multidisciplinary system: semantic ambiguity, absence of physical rigor and a lack of support for integration with simulation tools. To surpass these barriers, the Object management Group had developed a new iteration of the standard: SysML v2. this new version is not a simple update of the previous language, but a completely new approach to system engineering.

2.4.1 SysML v1: Overview and Limitations

> *"The Systems Modeling Language (SysML) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities"* [11]

According to [12], SysML consist of 4 different aspect of a system: requirements, behavior, structure and parametric, fig 2.13.

It takes its roots from UML 2 but with the objective of describe a wide range of systems (UML is commonly used for software). Without going too deep, the first iteration of SysML manifested a series of problems that limits its applicability, especially when trying to create an automatized framework with other tools:

- **Lack of a formal semantic** and rigorous syntax creates ambiguity, making models not executable. As a result, the meta-models are relegated as static documentation, loosing their value in a MBSE framework.

- **Lack of interoperability with other tools:** SysML v1 relies on data exchange based on XML, making hard the process of data extraction and exchange.

- **Lack of a native API** infrastructure makes the ecosystem dependent on vendor-specific extensions, impeding a direct integration with custom tools and framework.

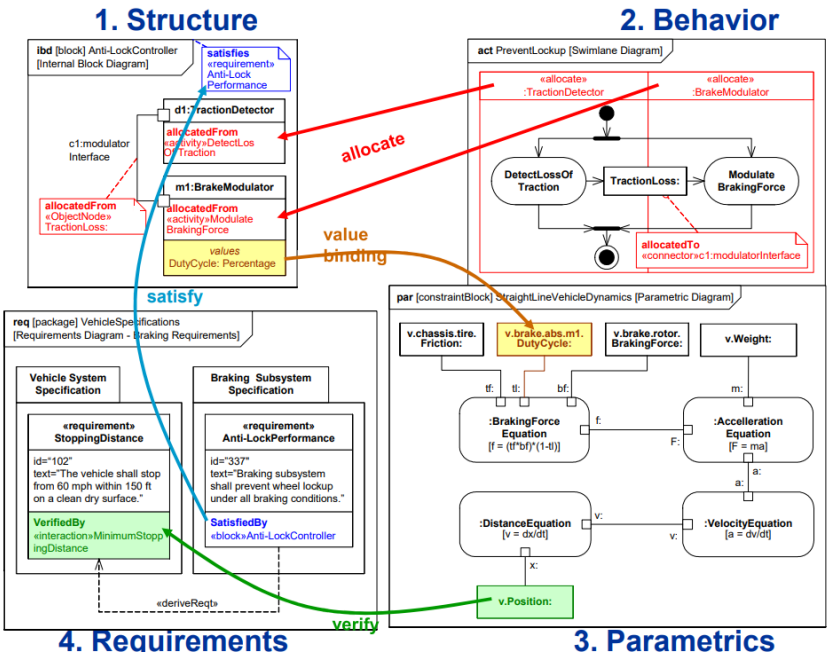

Figure 2.13: Four pillars of SysML

2.4.2 SysML v2: overview and advantages

> *"SysML is a general-purpose modeling language for modeling systems that is intended to facilitate a model-based systems engineering (MBSE) approach to engineer systems. It provides the capability to create and visualize models that represent many different aspects of a system. This includes representing the requirements, structure, and behavior of the system, and the specification of analysis cases and verification cases used to analyze and verify the system."* [13]

SysML v2, the successor of SysML, is currently under development and can be defined as a textual modeling language that refines and harmonizes the concepts of its predecessor and improve its syntax [14]. SysML v2 is not built on top of UML as a profile, instead, a new general-purpose modeling language has been designed to serve as foundation of domain-specific languages: the Kernel Modeling Language (KerML). The textual notation, enabled by KerML, can be used in parallel with the graphical notation, figure 2.15.

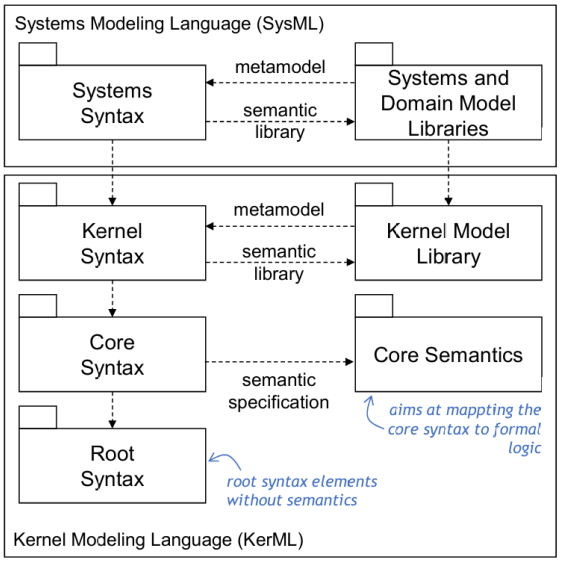

Figure 2.14: SysML v2 language architecture

The main advantages of SysML v2 over its predecessor are:

- **New metamodel based on a formal semantic:** SysML v2 is based on KerML, which provides a formal semantic. This declarative and ontological semantics interprets models in terms of classification, making it possible to mathematically verify the correctness of an interpretation against the model [16].

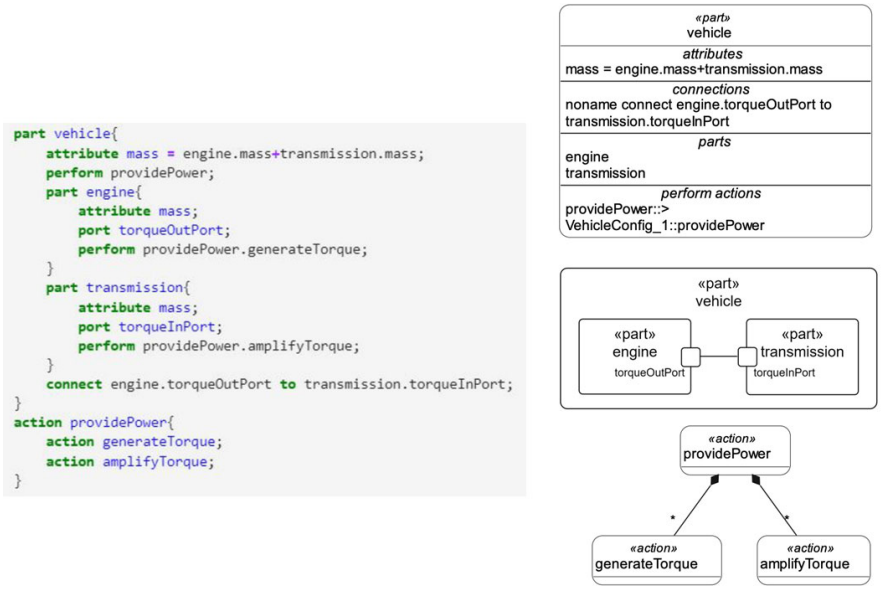

Figure 2.15: SysML v2 textual notation (left) and graphical (right) [15]

- **Textual paradigm:** This approach allows to a native, tool-independent exchange format, increasing interoperability with other tools.
- **Standardized API infrastructure:** the *System Modeling API* provides full programmatic access to model elements in a standard way, enabling direct interaction between tools without intermediate steps, fig 2.16.

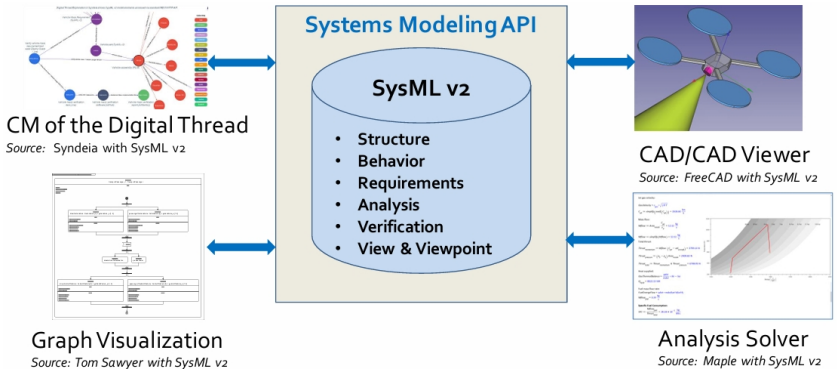

Figure 2.16: SysML v2 API capabilities [17]

SysML v2 has introduced new modeling concepts:

- **Requirements:** SysML v1 already had requirements but with the v2 version thay now can declare their subject, have attributes and define constraints to formally state the requirement.
- **Analysis and verification cases:** These new types can be used to define the calculation of a measure or the verification of a requirement, as well as a series of steps to derive one of its quantitative (analysis case) or qualitative (verification case) properties [16].
- **Annotations:** several new annotations can now be used to add information to model elements but without interfering with the semantic. An example is the annotation *metadata*, to capture any kind of meta-information, as well as tool-specific data.
- **Language extension and libraries:** Stereotypes and profiles are now replaced with semantic metadata to introduce user specific keywords. The functional meaning of these keywords is defined through the implicit specialization of existing standard library concepts. This architecture provides native and integrated language extensibility.

Chapter 3

Methodology and Implementation

This chapter outlines the methodology behind the proposed Virtual Verification workflow, designed to streamline the process of requirement verification within the V-model lifecycle. The goal is to demonstrate how we can leverage the principles of MBSE to effectively close the gap between system design and integration/testing.

To achieve this goal, an integrated workflow was developed that pairs SysMLv2 — used to formalize both the system architecture and its requirements — with Modelica, which handles the executable behavioral simulations.

When defining this methodology, it is necessary to consider the broader framework of this thesis. The ODE4HERA project focuses not only on MBSE and $V\&V$, but also on Multidisciplinary Design Optimization (MDO), Simulation Data Management (SDM), and Product Lifecycle Management (PLM). Consequently, the methodology must be designed for seamless integration with these disciplines and the existing tools developed at DLR. Specifically, it must support integration into larger MDO workflows, which require the automatic generation and execution of system architectures within a simulation environment such as Modelica.

This requirement for full automation highlights a critical limitation in standard translation processes. As illustrated in Figure 2.6, the generated Modelica model often acts merely as a structural template that requires manual extension with behavioral models. Because this manual integration occurs directly within the Modelica environment, it prevents the automatization required by the ODE4HERA framework. The methodology proposed in this chapter is specifically engineered to resolve this bottleneck.

The following sections will cover how the workflow operates step by step. The discussion begins with requirement definition and system design, followed by the

automated transition from descriptive SysMLv2 architectures to fully simulation-ready Modelica models. Finally, it covers the simulation execution, the extraction of results, and the automated feedback loop used to update the requirement verification status.

3.1 Methodology workflow

The methodology workflow is described in figure 3.1. As can be seen, there are 4 "domains" in which it was divided that we will briefly go through now, and with more detail in the next sub section. The process starts with the definition of the requirements, the starting point of any V-model approach. From these, a high-level representation of the system is derived. It must capture the main aspects and functions needed to fulfill the requirement. Here ends the high-level definition, and we start to go deeper adding more details and model based design informations.

Before doing so, if we want to have a process that can be automatized, we need to import in SysMLv2 the behavior of the system or, at least, a connection to it. This can be achieved by importing in SysMLv2 a portfolio of Modelica components that are generated before hand.

We can now define a Model Based (MBD) Architecture. This is simply a specialization of the high-level architecture, that incorporates information on the system behavior, being based on the imported components from Modelica. This architecture is the *solution* to all the requirements, in other words, it is *neutral*: it is a single architecture for all the requirements, it cannot be executed.

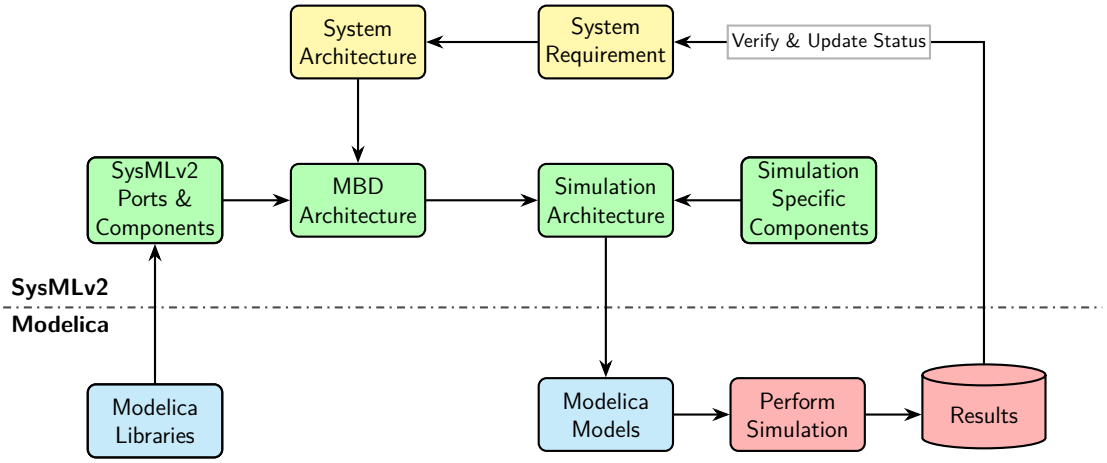

Figure 3.1: Proposed Methodology

To verify a requirement we need a test bench, a solution to the verification process. The solution is the simulation architecture, obtained by enriching the

MBD architecture with some simulation specific components. Information on the environment, working and boundaries conditions as well as tool specific settings are so added.

The enriched architecture has now everything that is needed to be translated in a simulation-ready model in Modelica. This is performed via a python tool. At this point we can run the simulations and use the results to verify the requirements. It must be noted that the information to do so are contained in the requirement definition, as it will be described in the next sub section.

As can be seen by looking at the methodology, everything relevant for requirement definition and architectural design is carried out completely in SysMLv2 while Modelica is mainly used for behavioral modeling and as simulation engine. Doing it this way, it is possible to blend this methodology in a bigger framework as mentioned before. Let's now break down the methodology step-by-step.

3.1.1 Requirement Definition

Requirements can be defined is SysMLv2 using the *requirement* type usage, in this thesis the ODE4HERA Ontology was used to add domain-specific attribute to the requirement. Some of the ontology can be seen in 3.1 where an example of requirement for the FCS is shown: the attributes *status*, *version*, *priority*, *implementationStatus* and *verificationStatus* are being used to define the status of the requirement, in particular, the last attribute is the one of most interest. The *verificationStatus* represents the progress in the process of verification of the requirement and its value is defined by *VVStatus*, an enumeration definition that can have multiple predefined value as, in this case, *NotStarted*.

```
requirement <'REQ-001'> MaxThresholdRequirement : Requirement {
        doc /* The target variable must not exceed the specified threshold. */

        subject Arch : TargetSystemGroup;

        attribute EnvTest : TestEnvironment {
            :>> ambientParameter = 1.0;
        }

        require constraint { TargetSystem.outputVariable <= 5.0 }

        :>> status = ReqStatus::Approved;
        :>> version = "1.0";
        :>> priority = ReqPriority::MustHave;
        :>> implementationStatus = ReqImplementationStatus::InProgress;
        :>> verificationStatus = VVStatus::NotStarted;
    }
```

Listing 3.1: SysML v2 General requirement definition

3.1.2 System Architecture

The Sysmlv2 document *System Architecture* is a high level representation of the architecture that defines the main component of the system, the attributes and, eventually, the connections. It is very important to note that this architecture does not go too far in the details: not all the components are visible at this level but only the most important ones, and they are represented with generic, functional names rather than specific physical implementations. For example, an actuation subsystem will appear simply as *Actuator* rather than *BrushlessMotor* or *HydraulicCylinder* — those details are introduced only at the MBD Architecture level.This level of abstraction keeps the system architecture neutral to implementation choices (EHA over EMA) without the need of updating it.

For the listing 3.2 an example of how this architecture may look is shown. We have all the *part definition* for the components like the *SubSystemA* and *SubSystemB*. Every component also has the main attributes useful to size the system, for example, for *SubSystemA* we have *parameterA1*, while for *SubSystemB* we have *parameterB1* and *parameterB2*. It is also present a higher level component, the *TargetSystemGroup*, that is composed by *subA* and *subB*. This is done to group components in a subsystem and lower the complexity.

Then we have two elements preceded by a keyword defined with a #. This is simply a *metadata* part of the ODE4HERA ontology that can be used to add specific information to a part. It is possible to see the *#architectureDesignSpace* part def *TargetSystemGroup*, where the design space of the system is defined, and the *#architecture* part *TargetSystem*, where it is instantiated and the attributes are assigned.

```
part def SubSystemA {
        attribute parameterA1 : Real;
    }

    part def SubSystemB {
        attribute parameterB1 : Real;
        attribute parameterB2 : Real;
    }

    #architectureDesignSpace part def TargetSystemGroup {
        attribute groupParameter : Real;
        attribute outputVariable : Real;

```

```
        part subA : SubSystemA;
        part subB : SubSystemB;
    }

    #architecture part TargetSystem : TargetSystemGroup {
        doc /* Instantiated System Architecture */

        part redefines subA {
            :» parameterA1 = 10.0;
        }
        part redefines subB {
            :» parameterB1 = 20.0;
            :» parameterB2 = 30.0;
        }
    }

    satisfy MaxThresholdRequirement by TargetSystem;
}
```

Listing 3.2: SysML v2 High level system architecture example.

3.1.3 Import of Modelica Libraries

It is important to distinguish between two categories of Modelica models: standard components from third-party libraries (e.g., *Modelica.Mechanics*) and custom-built components. While the import process is the same for both, keeping the namespace consistent is critical. To maintain traceability and avoid compilation errors, standard components should be organized in a SysML v2 package hierarchy that mirrors the original Modelica library, similar to the CATIA implementation (see Section 2.1.1).

The import process requires strict naming and structural rules. To guarantee a correct mapping and prevent simulation errors, the following rules must be followed:

- **Port Names:** SysML v2 ports cannot be named arbitrarily. They must exactly match the names of the physical Modelica connectors. If the names are different, the transformation engine will generate the routing equations (e.g., the `connect()` statements), but the simulation will fail because the physical port will not be found.
- **Model Path:** SysML v2 components must include a metadata attribute specifying their exact Modelica path. This metadata acts as a locator, telling the simulation engine where to find the mathematical model in the external library.
- **Attribute Binding:** System parameters must be imported into SysML v2 using their exact Modelica names. Whether defining a mechanical inertia or a

specific input current for a motor driver, identical naming ensures that the values assigned in SysML v2 are correctly passed to the physical model during initialization.

Importing a standalone interface is simpler: it only requires specifying the correct model path and keeping the exact original connector name.

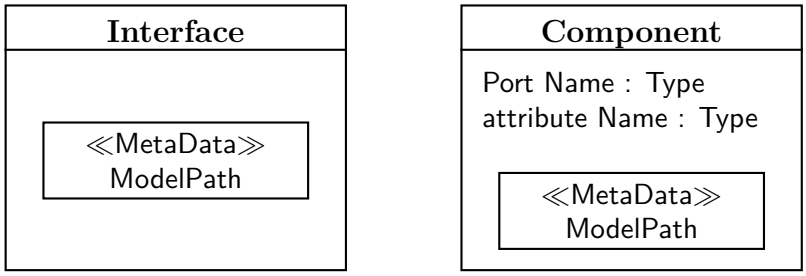

Figure 3.2: Schematic of imported component in SysMLv2

3.1.4 MBD Architecture

The Model Based Design Architecture is created after the system architecture is defined and all the components needed are imported from Modelica. This architecture has a higher level of detail compared to the previously defined one as the system is better characterized. The components better represent the real architecture, there can still be a little difference but it can also be 1:1 with the physical system.

A very important aspect is the usage of the components imported from Modelica. Nevertheless, this architecture can, depending on how the components are defined in Modelica, still be considered platform-neutral since there is yet no tool-specific characterization. It is important to say that, in the current work, in this document all the system specific components that were developed *Ad Hoc* are stored. These components are, more or less, the one defined by the system architect in the previous document. One example is proposed in the listing 3.3.

```
1 package Definitions {
2     #mbd_component part def MBD_ComponentA {
3         out port out_port_1 : GenericPortType;
4 
5         #mbd_parameter attribute parameterA1;
6 
7         @SimulationModelMetadata {
8             :» authoringTool = "OpenModelica";
```

```
            :» InternalModelPath = "GenericLibrary.ComponentA";
        }
    }
    #mbd_component part def MBD_ComponentB {
            in port in_port_1 : GenericPortType;

            #mbd_parameter attribute parameterB1;
            #mbd_parameter attribute parameterB2;

            @SimulationModelMetadata {
                :» authoringTool = "OpenModelica";
                :» InternalModelPath = "GenericLibrary.ComponentB";
            }
        }
}
```

Listing 3.3: SysML v2 MBD components.

Of a certain importance are the *#mbd_component* and *#mbd_parameter* metadata to identify this component as model based specific and the *@SimulationModelMetadata* to assign specific information as the *InternalModelPath* where the component can be found in Modelica. Also, all the attribute's and port's names must be equal to the one in Modelica otherwise they will not be recognized later. These components can also be stored in a separate document if too many.

At this point, it is possible to specialize the high-level architecture using the MBD components in order to obtain an MBD architecture. It must be said that this passage is not automated and must be done by the engineer. As already said, the two architecture are not 1:1 and some components may be added, see listing 3.4.

```
#mbd_architecture part MBD_System {
    :» systemArchitecture references Generic_Architecture::TargetSystem;

    #mbd_component part compA : Definitions::MBD_ComponentA {
        :» parameterA1 = 10.0;
    }
    #mbd_component part compB : Definitions::MBD_ComponentB {
     :» parameterB1 = 20.0;
        :» parameterB2 = 30.0;
    }

    // Connections
    connect compA.out_port_1 to compB.in_port_1;
    }
```

Listing 3.4: SysML v2 MBD architecture.

As it can be seen, the MBD architecture *MBD_System* is anticipated by the metadata *#mbd_architecture* and, inside of it, all the components and connections

are instantiated. About the MBD architecture, some aspects must be explained. The architecture itself references a specific part in the high-level representation through the keyword *references*: in this case, *MBD_System* references *TargetSystem*, defined in the *Generic_Architecture* package. For example, the component *compA* is typed as *Definitions::MBD_ComponentA* and its attribute *parameterA1* is redefined to *10.0*, while *compB* is typed as *Definitions::MBD_ComponentB* with *parameterB1* and *parameterB2* redefined to *20.0* and *30.0*, respectively. Also, it is important to note that attributes of the system or components can be redefined at this level, and that the two components are connected through the port *out_port_1* of *compA* to the port *in_port_1* of *compB*.

The big issue for the MBD architecture are the connections. Every port of every component must be connected or there is a high possibility that there will be an error in the simulation. As it can be seen, even for a small subsystem as the hydraulic actuation, the number of connection is very big and very had to manage since the port names must match exactly the one in Modelica. For this reason it is very important to put a lot of effort in the development of the models in Modelica, the better they are the easier the design in SysMLv2 will be. Effective models must have explicit names and, if possible, they must group as many ports as possible in to a single one. For instance, the poor management of the brushless motor led to having a port for each phase, resulting in a total of 9 ports. Nevertheless, it must be said the the textual notation of SysMLv2 make things harder and this problem will be surpassed once a graphical interface will be available.

3.1.5 Simulation Architecture

After the MBD architecture is ready, we must define the simulation architecture, always in SysMLv2. This architecture differs from the previous because simulation specific settings and components are here added. While the MBD architecture is one for every requirement, in the simulation architecture we want to have one architecture for each requirement, because every single one of them necessitates a different test bench with different environment and system condition. All the different settings and logic are defined with some custom metadata developed for this purpose and are shown in the following listings 3.5 and 3.6. A metadata with the name *SimulationSettings* was defined and it contains, as attributes, all the simulation parameters that can be set in Modelica.

```
1 metadata def SimulationSettings :> ExecutableModelMetadata {
2         attribute startTime : Real;
3         attribute stopTime : Real;
4         attribute numberOfIntervals : Integer;
5         attribute tolerance : Real;
6         attribute method : String;
```

```
        attribute outputFormat : String;
        attribute variableFilter : String [0..1];
    }
```

Listing 3.5: SysML v2 simulation settings.

Another one called *SimulationModelMetadata* serve the purpose of defining the type of simulation through the attribute *SimulationType*. This was done because, for what concerns the FCS, the requirements are usually given both in time and frequency domain, this attribute changes the type of simulation to be performed. *VerificationLogic* is simply an enumeration definition used later to determine which value must be taken to verify the requirement.

```
enum def VVLogic {
        doc /* Verification Logic */
        Max {
            doc /* Take maximum value */
        }
        Min {
            doc /* Take minimum value */
        }
        SteadyState {
            doc /* Take avarage final values */
        }
    }

    metadata def SimulationModelMetadata :> ExecutableModelMetadata {
            attribute InternalModelPath : ScalarValues::String [0..1]
            attribute SimulationType : String default "Time"; // Time -
    Frequency
            attribute VerificationLogic : VVLogic;
    }
```

Listing 3.6: SysML v2 simulation type.

Now, the simulation architecture can finally be created. For it to work we need to instantiate the MBD architecture and all the simulation specific components like inputs, sensors, boundary conditions, the connections and the simulation metadata. An example is shown in the listing 5.4: the simulation architecture is named *Generic_Test_Bench* and it contains the MBD architecture *SystemUnderSimulation* of the type previously defined in the other document. In this test bench we want to test the system for a generic requirement, this is defined in the documentation but also by the expression *satisfy MaxThresholdRequirement*. This is the link with the corresponding requirement that must be satisfied, it will be used later to trace it back. In the simulation settings we want to export the results in a csv file and only the variable of the speed sensor to avoid having a file too big.

```
#mbd_architecture part Generic_Test_Bench {
    doc /* Test bench for REQ-001 */

    @SimulationModelMetadata {
        :» authoringTool = "OpenModelica";
        :» SimulationType = "Time";
        :» VerificationLogic = VVLogic::Max;
    }

    #mbd_architecture part SystemUnderSimulation :> Generic_MBD::MBD_System;

    #mbd_component part abstractSensor : GenericLibrary::SensorBlock {
        :» enableOutput = "true";
    }
    #mbd_component part stimulusSource : GenericLibrary::SourceBlock {
        :» amplitude = 1.0;
    }

    // Test Connections
    connect stimulusSource.out to SystemUnderSimulation.compA.in_port_1;
    connect abstractSensor.in to SystemUnderSimulation.compB.out_port_1;

    satisfy MaxThresholdRequirement;

    @SimulationSettings {
        :» startTime = 0.0;
        :» stopTime = 10.0;
        :» numberOfIntervals = 500;
        :» tolerance = 1e-6;
        :» method = "dassl";
        :» outputFormat = "csv";
        :» variableFilter = "abstractSensor.outputVariable";
        }
}
```

Listing 3.7: SysML v2 simulation architecture.

3.1.6 Modelica Exported Models

From the simulation architecture, we have the capability to really automate the problem starting with the automatic creation of the Modelica model, with a very simple Python script that will be explained in section 3.2. The simulation architecture is translated to a model definition, while the MBD architecture is exploded in its components, as can be seen in the listing 3.8. All the attributes defined is SysMLv2 are exported as modifiers in Modelica to redefine the parameter value,

if not modifier is found, the default value is set. Unfortunately, a logic to organize the component within a graphical representation was non implemented, all the components are placed in the same point in space and no connections are drown.

```
model Generic_Test_Bench

  GenericLibrary.SourceBlock stimulusSource(amplitude=1.0);
  GenericLibrary.SensorBlock abstractSensor(enableOutput=true));
  GenericLibrary.ComponentA compA(parameterA1=10.0);
  GenericLibrary.ComponentB compB(parameterB1=20.0, parameterB2
   =30.0);

equation
  connect(stimulusSource.out, compA.in_port_1);
  connect(compA.out_port_1, compB.in_port_1);
  connect(compB.out_port_1, abstractSensor.in);

end Generic_Test_Bench;
```

Listing 3.8: Modelica exported model

3.1.7 Simulation and Results

Once the model is exported as a Modelica file (.mo), the process is almost over: with OMPythom it is possible to set the simulation settings and launch the simulation. At the end of it, the result data can be saved both in .mat or .csv file for post-processing and, at the end, to verify the requirement and update the status in SysMLv2. This process will be better explained in section 3.2.

At this point the models can be used to asses the fulfillment of the requirements. As it will be shown in chapter 4, for the Flight Control System we can have requirement both in the time domain and in frequency domain. For this reason it was necessary to integrate a strategy to perform both analysis.

For what concern the frequency analysis many solution were investigated but the most promising one was to linearize the model around the stability point (that must be specified in the requirement) and extract the steady-state matrices A, B, C and D. Then, from the matrices we can perform multiple analysis and generate bode plots. To accomplish that, we can use the scripting function *linearize* though OMPython, but more details will be given in the implementation subsection

3.2 Methodology Implementation

In order to add a certain level of automation to the methodology, Python code was developed. It must be said that the developed code does not aim to claim full automation of the entire process nor to be a final product ready to be distributed. It was developed as a proof-of-concept to assess the level of automatization that can be achieved and to discover the hidden limits of the methodology. Anyway, it is believed that a good level was reached and that even more can be done but, also, that not a fully automatized process, starting from the first phases in SysMLv2 may be possible to achieve and that, at some point, a physical person is needed.

Interfacing with SysMLv2 API was possible thanks to SysIDE Automator, the analysis and optimization of SysML v2 models, automation of modeling workflows, and building custom applications [18].

Since the SysIDE Abstract Syntax Tree (AST) for SysMLv2 is very complex and difficult to interact with, a custom library developed by DLR - DMS group called *sysmlv2-python-utils* was used. This library contains a variety of functions and methods to extract, manipulate and write in/to the AST.

Lastly, the same problem exists when writing the Modelica (.mo) file. To help interfacing with the AST, a python library called modelica-builder and developed by URBANopt was used [19]. The principal use case for this library is to load, modify using higher level abstracted methods, and then save the resulting file.

The structure of the code is simple and is shown in fig 3.3:

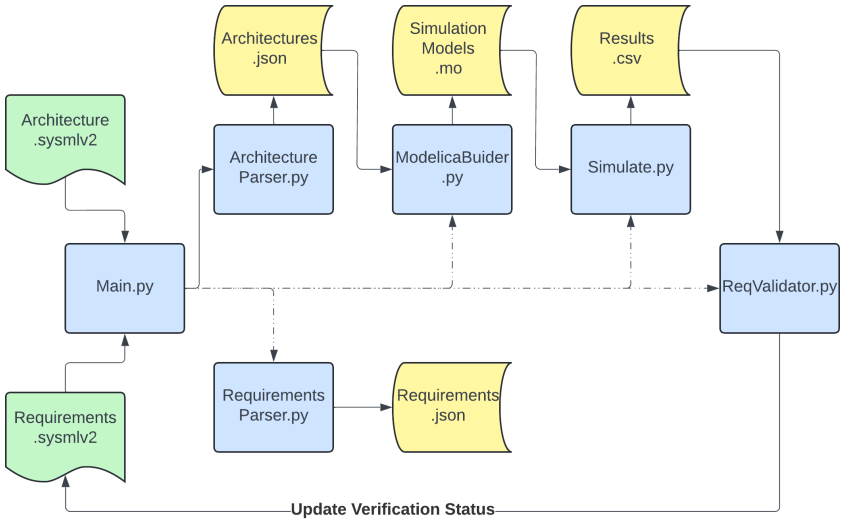

Figure 3.3: Python code architecture

A main script reads the two SysMLv2 documents containing the simulation architecture and the requirements. Then a SysMLv2 parser that uses SysIDE

automator and sysmlv2-python-utils extracts the data and stores them into a .json file as an intermediate step. At the same time, the requirements are parsed by an omonimum script that stores the data always in a json. The architecture.json is read by the script with modelica-builder that generates in output the .mo file. This is loaded and simulated by another script with OMPython and an output file (typically a .csv)is generated. Finally, a script performs the verification of the requirement and updates the *VVStatus*in the SysMLv2 document. In the following subsections, some key aspects of the implementation will be briefly explained.

3.2.1 Parsing and Model Generation

Parsing the SysMLv2 Abstract Syntax Tree is a crucial passage. SysIDE automator natively integrates a json serialization feature to export models in this format. Unfortunately, the exported file can be very complex and difficult to use to generate the Modelica dynamic model. To visualize the problem, a sample from an export will be shown in the listing 3.9:

```
{
  "@type": "OwningMembership",
  "@id": "0a680cbf-b149-5715-8acd-2ce5ab495768",
  "elementId": "0a680cbf-b149-5715-8acd-2ce5ab495768",
  "owningRelatedElement": { "@id": "152f4f90-24ad-43f0-9fc0-
    c58eccf681c5" },
  "memberElement": { "@id": "f20f3ad6-71e5-580d-bef8-22f8b3084bae"
    },
  "memberName": "JSON Export Example",
  "ownedRelatedElement": [
    { "@id": "f20f3ad6-71e5-580d-bef8-22f8b3084bae" }
  ]
}
```

Listing 3.9: Exported json with SysIDE automator

Having this problem, a custom SysMLv2 parser was developed. The script was designed to accurately parse architectures, components and ports and save them in an organized way capturing all the necessary data, as can be observed in the listing 3.10. It is important to note that the redefined attributes in SysMLv2 are saved in the json file under the attribute *modifiers*. Also, the connections are saved taking the two ends.

```
{
  "package": "Simulation_Architecture",
  "architectures": [
    {
```

```
5       "name": "Generic_Test_Bench",
6       "qualified_name": "Simulation_Architecture::
    Generic_Test_Bench",
7       "Requirements": {
8         "name": "MaxThresholdRequirement",
9         "short_name": "REQ-001"
10      },
11      "components": [
12        {
13          "name": "compA",
14          "qualified_name": "Generic_MBD::Generic_Test_Bench::
    compA",
15          "type": "GenericLibrary.ComponentA",
16          "modifiers": [
17            {
18              "name": "parameterA1",
19              "value": 10.0
20            }
21          ]
22        },
23        {
24          "name": "compB",
25          "qualified_name": "Generic_MBD::Generic_Test_Bench::
    compB",
26          "type": "GenericLibrary.ComponentB",
27          "modifiers": [
28            {
29              "name": "parameterB1",
30              "value": 20.0
31            },
32            {
33              "name": "parameterB2",
34              "value": 30.0
35            }
36          ]
37        },
38        ...
39
40      ],
41      "connections": [
42        {
43          "ends": [
44            "stimulusSource.out",
45            "compA.in_port_1"
46          ]
47        }
48        ...
49
50          ]
```

```
        }
      ],
      "variables": null,
      "attributes": [],
      "constraints": [],
      "simulation_metadata": {
        "authoringTool": "OpenModelica",
        "SimulationType": "Time",
        "VerificationLogic": "Max"
      },
      "simulation_settings": {
        "startTime": 0.0,
        "stopTime": 10.0,
        "numberOfIntervals": 500,
        "tolerance": 1e-06,
        "method": "dassl",
        "outputFormat": "csv",
        "variableFilter": "abstractSensor.outputVariable"
      }
    }
  ]
}
```

Listing 3.10: Proposed exported json

To extract basic information as the name, the qualified name, the value of an attribute or the type of an usage we can simply access the data with the SysIDE API, but for more complex task ad-hoc function were created and with the support of the sysmlv2-python-utils, an example is shown in listing 3.11.

```
def _extract_sim_settings(self, element: syside.Element):
        """
        Parse SimulationSettings
        """
        metadata = {}
        for node in self._collect_children(element):
            if isinstance(node, syside.MetadataUsage) and
   node.metaclass.name == "SimulationSettings" :
                for feature in node.owned_features:
                    metadata[feature.name] =
   self.model_helper.parse_feature_value(feature)
        return metadata
```

Listing 3.11: Example of custom function for data extraction

Once the json file has been created, the *modelica_generator* import this document and generates the *.mo* file. The process is very simple thanks to the *modelicaBuilder* library for python: all the different simulation architectures are

converted to Modelica models and stored inside an external package. An example of how the library works is shown in the listing 3.12: with the function *builder.insert_component* we can allocate a part specifying the name, the type (which is the Modelica path to the component in the library), the modifiers (redefined attributes) and the annotation containing, in this case, the placement for the graphical representation.

```
# Insert Components
    for component in model_data.get('components'):
        comp_type = component.get('type')
        comp_name = component.get('name')

        modifiers = {
            mod['name']: mod['value']
            for mod in component.get('modifiers')
        }

        builder.insert_component(comp_type, comp_name,
   modifications=modifiers, annotations=placement)
```

Listing 3.12: Inserting components in the Modelica AST

3.2.2 Simulation

As already mentioned, when executing a model, we mainly have two options: *simulate* or *linearize*. We want to simulate a model when assessing requirement in the time domain, ex: we want to see the maximum actuated velocity or the position error while following a command. On the other hand, we want to linearize the system around a stability point when we need to know, for example, the bode response of the actuator. We already covered how we distinguish this, but let's now see how we can perform this two different analysis.

Starting with the simulation, we can send an expression to the OMC server that calls the scripting function *simulate*. This function allows us to set the parameters of the simulation, decide the output format, the variables to export and other properties that are shown in the listing 3.13:

```
function simulate
  input TypeName className;
  input Real startTime = 0;
  input Real stopTime = 1.0;
  input Integer numberOfIntervals = 500;
  input Real tolerance = 1e-6;
  input String method = dassl;
```

```
  input String fileNamePrefix = "<default>";
  input String options = "<default>";
  input String outputFormat = "csv";
  input String variableFilter = ".*";
  input String cflags = "<default>";
  input String simflags = "<default>";
  output SimulationResult simulationResults;
  record SimulationResult
    String resultFile;
    String simulationOptions;
    String messages;
    Real timeFrontend;
    Real timeBackend;
    Real timeSimCode;
    Real timeTemplates;
    Real timeCompile;
    Real timeSimulation;
    Real timeTotal;
  end SimulationResult;
end simulate;
```

Listing 3.13: Modelica simulate function [20]

The same principle applies for the *linearize* command. The only difference is that to set the output format we need to use the *SetCommandLineOptions* command. An example is shown in the code 3.14 where we set as dump language python, we initialize the model and then linearize it at stopTime (point in time where linearization is performed). The A, B, C and D matrices are saved in *lin_data*.

```
# Linearize model
    omc.sendExpression(f"setCommandLineOptions(
    \"--linearizationDumpLanguage=python\")")
    # it is possible to change language (matlab, python ...)

    omc.sendExpression(f"setCommandLineOptions(
    \"-d=initialization\")")

    lin_data = omc.sendExpression(f'linearize({lin_model},
   stopTime={sim_settings["stopTime"]})')
```

Listing 3.14: Linearize a model with OMPython

Chapter 4

Application Case - The Flight Control System

The present chapter introduces the Flight Control System of a generic commercial aircraft, which will be used as application case for the verification methodology proposed in Chapter 3 and demonstrated in 5. The chapter is organized in two parts: the first one provides a technical background to understand the system, focusing mainly on the actuation technologies more considered for future generation aircraft, and what performance requirements the actuators must met. The second part documents the dynamic simulation models developed in OpenModelica, covering modeling choices, library selection, model development and preliminary validation against reference data.

4.1 FCS Architecture

The Flight Control System is responsible for the actuation of pilot commands into deflection of the aircraft aerodynamic surfaces, controlling the attitude and trajectory. In a modern commercial aircraft the FCS operates in a fly-by-wire configuration. The pilot inputs are measured by sensors, processed by the Flight Control Computer (FCC) and transmitted as electrical commands to the individual actuator mounted on each surface. The controlled surface can be divided in primary and secondary. In this work the focus was given to the primary control surface of the elevators, for which sufficient geometric data and reference results are available in literature to size and validate the actuator models.

Each actuator receives a position or rate command from the FCC and drives the surface accordingly, while feeding back measured position and velocity data through a sensor. The control loop is closed by the Actuator Control Electronics (ACE) unit. Understanding the actuation technology is therefore fundamental.

More Electric Actuators: Conventional fly-by-wire actuators rely on a centralized hydraulic circuit routed to power the flight control actuators. This approach is widely used and well-understood, but it carries significant penalties in weight, maintenance and power efficiency. The more electric aircraft (MEA) aims to replace hydraulic and pneumatic power distribution with a centralized electrical bus and electrically powered actuators locally placed to each surface.

Two main technologies have emerged: the Electro-Mechanical Actuator (EMA) and the Electro-Hydrostatic Actuator (EHA). The difference basically on how the moving force is generated and how they behave under failure.

4.1.1 Electro-Mechanical Actuators (EMA)

Electromechanical actuators can be either linear or rotational but to drive control surfaces the first type is the most commonly used. A gearbox reduce the rotation speed in output of the motor's rotor before a screwball convert the rotational motion into linear.

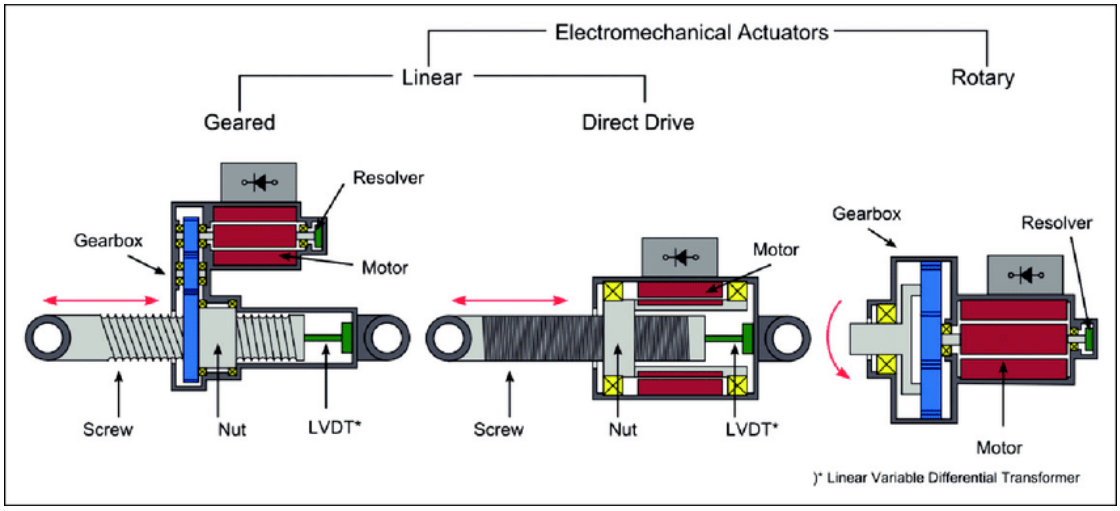

Figure 4.1: different types of EMAs for actuators [21]

The history of EMAs in flight control dates back to the 1970s, when early experimental installations appeared on spacecraft including the Space Shuttle. Ground and flight testing campaigns through the 1980s progressively built confidence in the technology, culminating in a demonstration programme on the C-141 aileron. A subsequent effort on the A320 right aileron accumulated significant flight hours and cycles, reaching TRL6 [21]. Despite this progress, no production aircraft currently relies on EMAs for primary flight control. The reason why EMAs have not yet reached primary flight controls lies in their failure behavior: a mechanical jam locks

the actuator in position, and since the surface is rigidly coupled to the actuator shaft, no redundant unit can backdrive it. This makes jamming a potentially catastrophic failure mode, fundamentally different from what is seen in hydraulic systems. For this reason EMAs are generally used in less safety-critical applications such as flaps, slats, spoilers, and trim horizontal stabilizers where jamming is not a catastrophic failure mode. As today EMAs are already in service for four of the 14 spoilers of the Boeing 787 and for the trimmable horizontal stabilizer of the Airbus A350.

In figure 4.2 is shown the typical architecture of an EMA actuator. The ACE receives two inputs: the position command issued by the FCC and the position feedback provided by the LVDT sensor. Based on these signals, the ACE applies the control law and generates as output a driving signal to the PDE. This then modulates the current supplied to the electric motor, typically a BLDC or a PMSM, in accordance with the ACE command. Finally, the rotary motion is produced by the motor shaft and converted into linear displacement through a gearbox and a screw jack assembly.

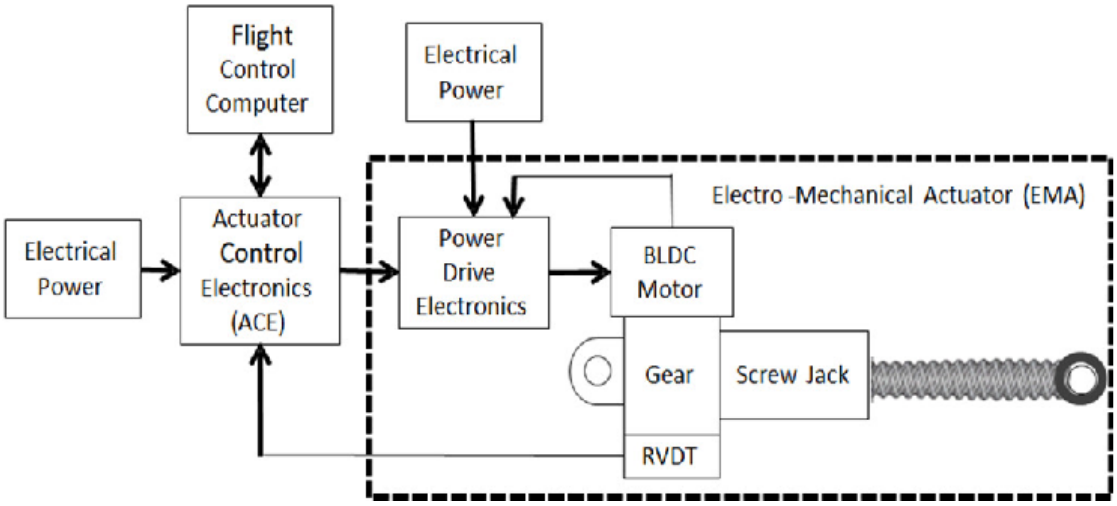

Figure 4.2: Electromechanical actuator scheme [22]

4.1.2 Electro-Hydrostatic Actuators (EHA)

An Electro-Hydrostatic Actuator can be of two different types: one with a constant speed motor and a variable displacement pump (EHA-VP) and another with a variable speed motor and a fixed displacement pump (EHA-FP). The EHA-FP configuration has emerged as the preferred solution for flight control applications. The reason is straightforward: the dominant operating condition during flight is one of sustained load at near-zero velocity, a regime in which a variable-displacement pump with a fixed-speed motor would keep the motor running continuously at partial load, an inefficient and thermally demanding condition. The EHA-FP avoids this by throttling motor speed directly, consuming power only in proportion to the actual demand [23, 24]. Still, this type of motor is preferable for certain application since its simple to control and due to the low inertia of the swashplate of the pump it has a higher frequency response.

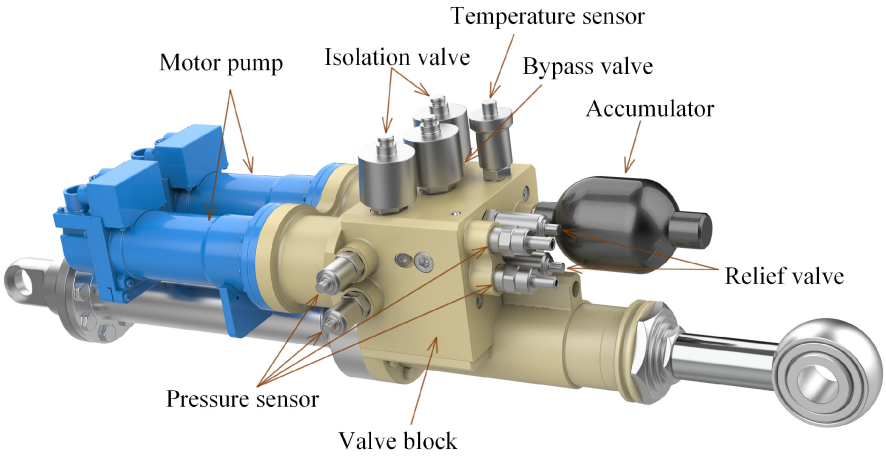

Figure 4.3: Electro-Hydrostatic Actuator [25]

A more detailed representation of an EHA-FP architecture can be found in figure 4.4. The electric motor is driven by a controller analogous to the one described in figure 4.2. In this configuration, the motor drives a fixed displacement pump which is fed by an accumulator via two check valves. Two relief valve are used as overpressure protection: if the system pressure exceed the design limit, they open and direct the flow back to the accumulator. Additionally, a bypass valve is included to isolate the motor-pump assembly in the event of a failure in one of these components.

The key advantage of EHAs over EMAs lies in their failure behavior. While a jammed EMA locks the surface completely — preventing any redundant actuator from moving it — an EHA with a bypass valve simply transitions into a passive damper mode, leaving the surface controllable by the remaining healthy units [27,

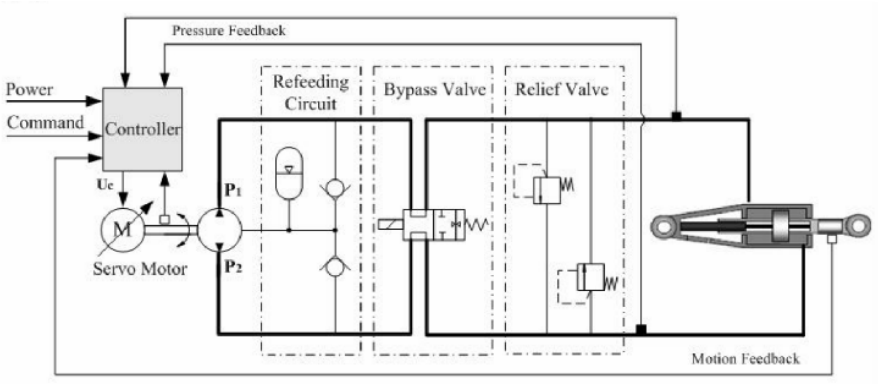

Figure 4.4: EHA-FP schematic [26]

25]. In figure 4.5 it is possible to see a fault-tolerant architecture with two valves (Valve 1 and Valve 2) that are used to isolate the motor-pump from the hydraulic cylinder.

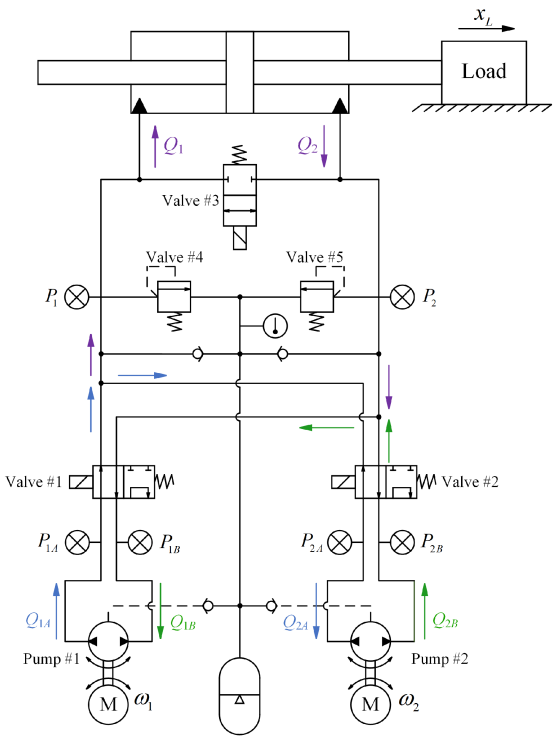

Figure 4.5: Double redundant EHA [25]

This makes the certification process a lot easier and EHAs can already be found on the A380 which is the first aircraft to feature fly-by-wire actuators on primary flight controls. As shown in figure 4.6 EHAs are installed on inboard ailerons, mid ailerons and elevators and backup EHA (EBHAs) are used on the rudder and spoilers. By removing the hydraulic line and replacing it with two electric systems Airbus could achieve a mass reduction of around 450 kg.

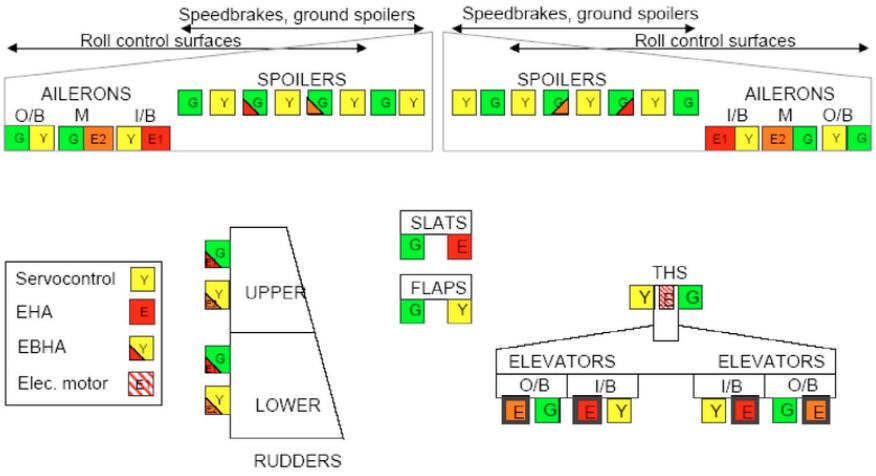

Figure 4.6: Flight control surfaces of the A380 [28]

4.2 Specification for stationary and dynamic performance

The requirements for the Flight Control System can be divided in two categories: Functional requirements due to safety aspects or performance requirements. Due to the nature of this work only performance requirements will be investigated being the ones prone to be analysed with dynamic simulation in OpenModelica.

4.2.1 Performance Requirements

Performance requirements can be divided in two categories: stationary and dynamic. The society of Automotive engineers has laid down a frequently quoted standard for actuator specifications with ARP 1281, "General Specifications for Hydraulic Power Operated Aircraft Flight Control System". Mostly, stationary requirements can be described by three characteristic operating points, as described in [27] and seen in figure 4.7 a), which are the following:

- The stall load $F_{stall} =: F_0$
- The max mechanical power $P_{mech,max} = max(F \cdot \dot{x})$ to be provided by the actuator, with the load F_1 and the speed $\dot{x_1}$
- The max speed with max corresponding load $(\dot{x_2}, F_2)$, or no load

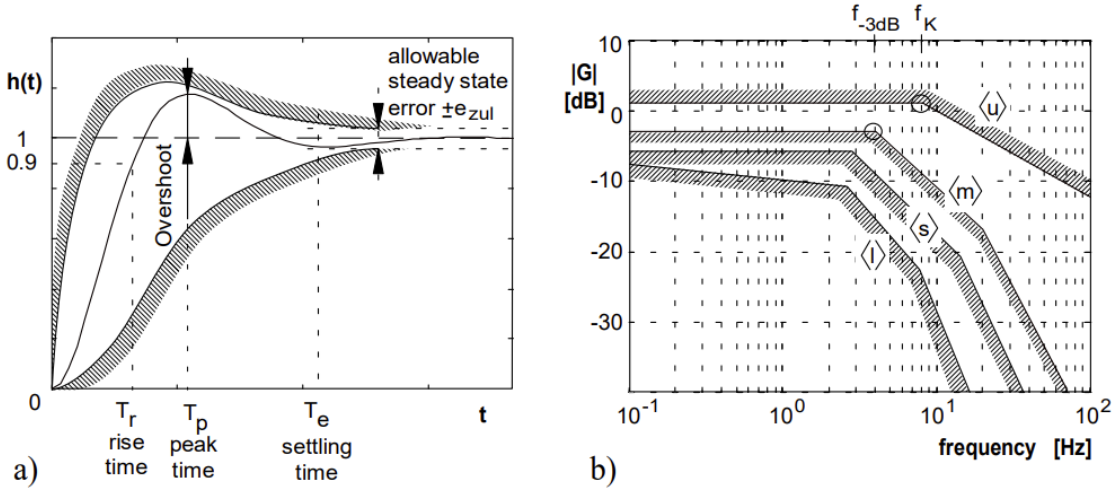

Figure 4.7: a) Step response, b) Frequency response [27]

Performance requirements are typically expressed as tolerance bands on the system response. For static performance, key metrics such as rise time, overshoot, settling time, and steady-state error can be easily assessed.

The frequency-domain behavior is usually specified, for a military aircraft, by means of bounds in the bode diagram as can be seen in figure 4.7 b). The bounds are generally determined by the maximum required command frequency, which depends on the control surface. Since ARP 1281 is not publicly available, the relevant requirements such as gain and phase margin are extracted from [29] and reported in 4.1, 4.2.

Table 4.1: Key dynamic characteristics for preliminary actuator design

Parameter	**Requirement**
Gain margin	6 dB (without aerodynamic damping)
Phase margin	45°
Frequency response	An input of 2 . . . 5% amplitude and an input of 10 . . . 25% amplitude shall be used. The output at these inputs shall be within specified maximum and minimum limits.
Transient response	A step input of 5%, 10% and 25% of total amplitude, for both loaded and unloaded conditions shall be used.

Table 4.2: Dynamic response characteristics for civil aircraft

Frequency	**Amplitude Ratio**	**Phase Shift**
0.5 Hz	-3 dB . . . 2 dB	$< 20°$
2.0 Hz	$\leq$ 2 dB	$< 50°$

As shown in the table 4.2 it is a common practice for civil aircraft to specify amplitude ratio and phase shift only at discrete input frequencies. Two further important specifications are the bandwidth the gain-drop. The bandwidth is defined as the frequency where the gain falls to -3dB, it is a critical parameter because, beyond this frequency, inputs become heavily attenuated and are effectively filtered out by the system. The gain-drop is defined as the slope of the gain curve above the bandwidth frequency and by demanding a certain decrease of gain the excitation of structural vibrations can be suppressed. An example of performance requirements for a light military jet trainer can be found at [30] and is reported in table 4.3, comprehensive of some basic requirements for EMA and EHA actuators.

Table 4.3: General Requirements for EHA and EMA actuator sizing

Parameter	Requirement / Value
General Sizing (Common)	
Equivalent load mass	50 kg
Stiffness (link to aircraft structure)	20 kN/mm
Stiffness (link to control surface)	8 kN/mm
Working stroke	±50 mm
Maximum no-load velocity	150 mm/s
Stall force	30 kN
Position response bandwidth	> 4 Hz
Position response steady-state error	$< 0.1\%$
Motor & Control (Common to EHA/EMA)	
Motor type	3-phase AC synchronous
Motor inertia	< 0.001 kg m^2
Motor maximum current	< 40 A_{rms}
Motor speed response bandwidth	> 40 Hz
Current response bandwidth	> 400 Hz
Actuator friction force	$< 5\%$ stall force
EHA Specific (Electro-Hydrostatic)	
Hydraulic fluid	MIL-H-5606B
Maximum working pressure	210 bar
Reservoir pressure	10 bar
Reservoir capacity	< 1 litres
Maximum internal leakage	$< 1\%$ (no-load max flow)
Pump inertia	< 0.001 kg m^2
EMA Specific (Electro-Mechanical)	
Mechanical transmission type	Ball-screw jack

4.2.2 Airbus A320 example requirements

More specific requirements for the Airbus A320 were found in [31] were a Application Case for flight control system sizing and hinge moment estimation is presented. Very useful for the work are the results in table 4.4 (mixed with some more requirements found at [23]) where some static requirements of the A320 are reported such as the stall load or alternatively the maximum hinge moment and the arm of the actuator for different control surfaces.

As mentioned in [31] and [23] these requirements for the FCS are estimates evaluated using approximate methods. Specifically, the maximum hinge moment can be estimated using the following equation:

$$M_{hinge} = q \cdot S_w \cdot c \cdot C_h \tag{4.1}$$

Where C_h is the hinge coefficient and can be decomposed in different terms:

$$C_h = C_{h_0} + C_{h_\alpha} \cdot \alpha + C_{h_\delta} \cdot \delta + C_{h_{\delta_t}} \cdot \delta_t \tag{4.2}$$

And the coefficients are evaluated following Roskam's method [32].

Table 4.4: A320 FCS data and requirements

Surface	**Aileron**	**Elevator**	**Rudder**
Actuator Config.	2 total (1 active)	2 total (1 active)	3 total (active)
Geometry & Loads			
Deflection Angle	$\pm 25°$	$+30°/-17°$	$\pm 25°$
Lever Arm [m]	0.047	0.071	0.116
Stroke [mm]	44	60	110
Actuator only Requirements			
Max Force / Stall Load [kN]	48.0	27.7	44.3
No Load Rate [mm/s]	90	60	110
Control surface Requirements			
Maximum Hinge Moment [Nm]	2115	1990	5105
No Load Rate [deg/sec]	60	60	60

4.3 Dynamic Simulation Models

The requirements defined in the previous section provide the specification against which the simulation models must be built. This section documents the development process of these models in OpenModelica, covering many different physical domains: the electric motor, the control surface kinematics and the hydraulic circuit.

Before starting with the description of the models, it must be clarified the modeling approach. Only the elevator surface is modeled because it has sufficient geometric and aerodynamic data available in the literature [23]. Extending these models to other aerodynamic surfaces is technically straightforward, following the same process. The models are then developed at a "system level", which means that the goal is to capture "well enough" the dynamic behavior of the system, not to reproduce every physical detail. The intended use of the models is to validate the proposed automated $V\&V$ methodology. Nevertheless, a strong effort was putted for developing these models, because their quality also determines the validity of the virtual verification of requirements.

Library Selection

Before starting with the modeling, the right Modelica libraries for each physical domain had to be selected. The choices made are explained below.

- **Mechanical domain - PlanarMechanics vs. MultiBody:** The control surface kinematics is essentially a planar mechanism, where a push-rod drives the surface rotating around a fixed hinge. While the standard `Modelica.MultiBody` library could in principle be used, planar mechanisms with kinematic loops often cause numerical problems in that library. For this reason the open-source `PlanarMechanics` library was chosen since it is developed specifically for this type of geometry.

- **Hydraulic domain - OpenHydraulics vs. ThermofluidStream:** Both libraries have all the components needed to model the EHA-FP hydraulic circuit and are very similar to each other. `OpenHydraulics` was preferred because it requires less thermal input data, which are difficult to find in public sources for this type of components. It is worth noting that switching to `ThermofluidStream` would be straightforward since the components are basically the same.

- **Electric motor domain - custom model:** This was the most challenging domain. The available open-source libraries for BLDC and PMSM motors, such as `openIMDML`, are either too detailed for system-level simulation or only supported in Dymola. The `Modelica.Electrical` library has some simple

motor models but they are not easy to parametrize for the specific motors used in this work. Not having a suitable option available, a custom BLDC average model was developed.

4.3.1 Literature Models

Since the purpose of the present work is only partly the development of flight control system models in OpenModelica, existing open-source models were investigated in order to save time and to have some reference cases against which the validity of the model could be assessed. Unfortunately, not many models were found, with the exception of one presented at ICAS 2010 [30]. In this work, the ElectroHydraulic actuator is modeled using HyLib library for Dymola environment which has very similar components to OpenHydraulic and so the model proposed in fig 4.8 was used as a reference. For the ElectroMechanical actuator the architecture is basically the same but instead of the hydraulic lines a gearbox and a ballscrew are used to drive the surface.

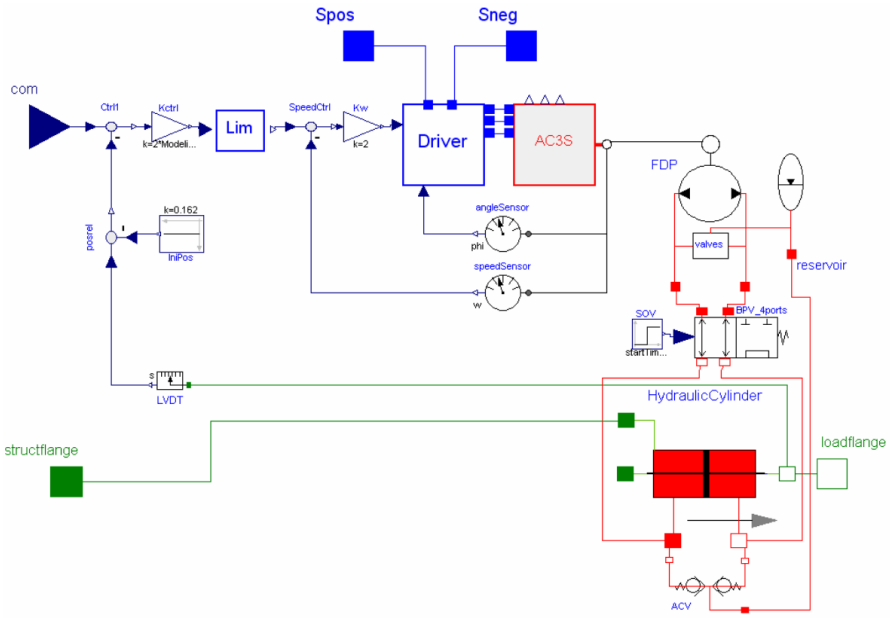

Figure 4.8: Di Rito [30] EHA Model

Always from [30] a Modelica model for the mechanical interface with the structure can be found, fig 4.9, and being compatible with the elevator kinematic it was used as a reference

Of great help was the work of Michael Anthony Cooper [23], an extensive work on actuator modeling and sizing to simulate the power consumption of an A320

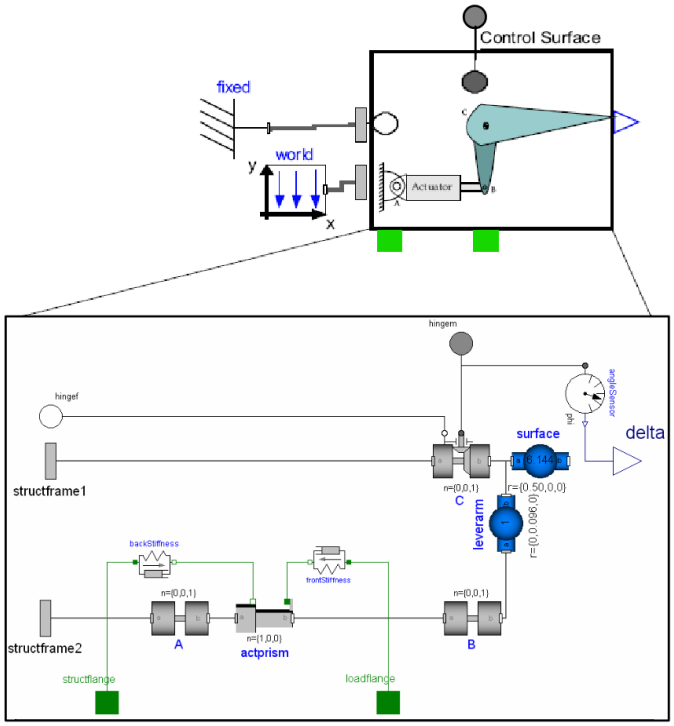

Figure 4.9: Di Rito mechanical interface [30]

during a reference flight. This work was of great help as it contains a lot of result data, characteristic curves and dynamic response of proposed A320 EMAs and EHAs models and it was vital to validate the models that will be proposed in this work. Another important work was the publication [24] about hybrid actuation in primary flight control systems. As Cooper PhD Thesis, also this work contains requirements, sizing parameters, characteristic curves, dynamic response for A320 EMAs and EHAs and was exploited together with Cooper's as a second benchmark.

4.3.2 Electric Motor

The industrial standard for electric motors is represented by the brushless DC motor (BLDC) and the permanent magnet synchronous motor (PMSM) but in this work more attention was given to the BLDC due to beeing a DC motor. The BLDC motor has become the preferred choice in aerospace actuation thanks to its high power density, elevated operating speeds, and the inherently low maintenance that comes from replacing mechanical brush commutation with electronic switching [33]. The controller switch DC currents to the motor windings, producing magnetic fields that effectively rotate in space and which the permanent magnet rotor follows. By adjusting the amplitude of the phase current it is possible to control speed and torque. To adjust the current flowing through the phases an inverter circuitry is usually implemented and the modulation is achieved with pulse width modulation (PWM). This technique allows to adjust the current by effectively controlling the voltage by lengthening or reducing the pulse time (typically referred as Duty Cycle). Increasing the duty cycle has the same effect as raising the voltage. This modulating technique requires high frequency switching (20 kHz) in order to properly control the motor but such a high frequency would kill a simulation making it too slow, at least, for the purpose of this work. With all the above considerations, an "average model", that is, a Brushless DC motor but with a simplified controller to avoid the 6-step commutation logic and PWM modulation that slows down the simulation, was developed. To assess the validity of the model a comparison of the key characteristics with a real motor is necessary. To do so, The Parker M series of BLDC motor was considered due to the high number of data available, in particular the Parker M1053K an M1054K [34] were considered and developed in this work, some of the key parameters are reported in table 4.5. The choice ended on these models also because in Cooper's thesis [23] these models were used and so his results were used as a benchmark to asses the models validity.

BLDC average model

To develop the mathematical model [33] was adopted as reference. The three phases star connected BLDC motor equivalent circuit can be observed in fig 4.10 and can

Table 4.5: Technical specifications for Parker M1053K and M1054K

Parameter	**M1053K**	**M1054K**
Torque Specifications		
Continuous Stall Torque (T_{cs}) [1]	8.3 Nm	10.2 Nm
Peak Torque (T_{pk})	24.9 Nm	30.6 Nm
Current Specifications		
Continuous Stall Current (I_{cs}) [1]	14.37 A (RMS)	16.76 A (RMS)
Continuous Stall Current (I_{cs}) [1,3]	20.32 A (Peak)	23.70 A (Peak)
Peak Current (I_{pk})	61.0 A (Peak)	71.1 A (Peak)
Rated Specifications		
Rated Speed (ω_r)	5000 rpm	5000 rpm
Rated Shaft Power (P_{out})	3739 W	4037 W
Rated Torque (T_{rated})	7.14 Nm	7.71 Nm
Rated Current (I_{rated}) [1]	12.60 A (RMS)	12.97 A (RMS)
Max Bus Voltage	560 VDC	560 VDC
Winding Constants		
K_t (Sine) [2]	0.576 Nm/A (RMS)	0.606 Nm/A (RMS)
K_t (Trap) [2,3]	0.470 Nm/A	0.495 Nm/A
Voltage Constant (K_e) [2,3]	0.470 V/rad/s	0.495 V/rad/s
Resistance (R) [2]	0.48 Ω	0.36 Ω
Inductance (L) [4]	2.0 mH	1.67 mH
Mechanical Data		
Rotor Inertia (J)	4.8×10^{-4} kg·m^2	6.2×10^{-4} kg·m^2
Motor Weight	6.6 kg	8.4 kg
Viscous Damping (B)	1.62×10^{-4} Nm/(rad/s)	3.03×10^{-4} Nm/(rad/s)
Static Friction (T_f)	0.042 Nm	0.042 Nm

Notes:
[1] @ 40°C ambient, derate phase currents and torques by 7%.
[2] Measured Line-to-Line, +/- 10%.
[3] Value is measured peak of sine wave.
[4] Inductance is measured Line-to-Line, +/- 30%.

be described by the following equations:

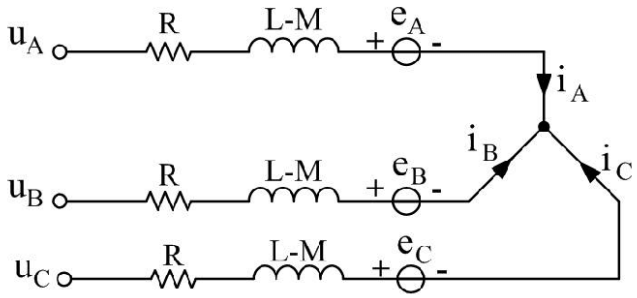

Figure 4.10: BLDC equivalent circuit

$$v_{ab} = R(i_a - i_b) + L\frac{d}{dt}(i_a - i_b) + e_a - e_b \tag{4.3}$$

$$v_{bc} = R(i_b - i_c) + L\frac{d}{dt}(i_b - i_c) + e_b - e_c \tag{4.4}$$

$$v_{ca} = R(i_c - i_a) + L\frac{d}{dt}(i_c - i_a) + e_c - e_a \tag{4.5}$$

$$T_e = k_f\omega_m + J\frac{d\omega_m}{dt} + T_L \tag{4.6}$$

Where v, i and e denote the line-to-line voltages, currents and back-emf respectively, in teh three phases a, b and c. Also the resistance R and inductance L are line-to-line while T_e and T_L are the electrical torque and the load torque. J is the rotor inertia, k_f is a friction parameter and w_m the rotor speed. Back-emf and electrical torque can be expressed as:

$$e_a = \frac{k_e}{2}\omega_m F(\theta_e) \tag{4.7}$$

$$e_b = \frac{k_e}{2}\omega_m F\left(\theta_e - \frac{2\pi}{3}\right) \tag{4.8}$$

$$e_c = \frac{k_e}{2}\omega_m F\left(\theta_e - \frac{4\pi}{3}\right) \tag{4.9}$$

$$T_e = \frac{k_t}{2}\left[F(\theta_e)i_a + F\left(\theta_e - \frac{2\pi}{3}\right)i_b + F\left(\theta_e + \frac{2\pi}{3}\right)i_c\right] \tag{4.10}$$

with K_e and K_t are the back-emf constant and the torque constant. The rotor angle θ_e is equal to the rotor angle times the number of pole pairs ($\theta_e = \frac{p}{2}\theta_m$. The function F gives the trapezoidal waveform to the back-emf and can be written as:

$$F(\theta_e) = \begin{cases} 1, & 0 \leq \theta_e < \frac{2\pi}{3} \\ 1 - \frac{6}{\pi}(\theta_e - \frac{2\pi}{3}) & \frac{2\pi}{3} \leq \theta_e < \pi \\ -1 & \pi \leq \theta_e < \frac{5\pi}{3} \\ -1 + \frac{6}{\pi}(\theta_e - \frac{5\pi}{3}) & \frac{5\pi}{3} \leq \theta_e < 2\pi \end{cases} . \tag{4.11}$$

The equation were implemented in Modelica but with a slight variation (to exploit its potential and to make the code easier to read), the star point equation has been separated and the trapezoidal function has been implemented in a separate model 4.1.

```
model BLDCMotor

  // Variables

equation
  // Interface mapping
  v_a = pin_a.v;
  v_b = pin_b.v;
  v_c = pin_c.v;
  i_a = pin_a.i;
  i_b = pin_b.i;
  i_c = pin_c.i;
  w_mech = der(flange.phi);
  theta_elec = p * flange.phi;

  // Trapezoidal Shapes
  shape_a = BLDC.TrapFunc(theta_elec);
  shape_b = BLDC.TrapFunc(theta_elec - 2*pi/3);
  shape_c = BLDC.TrapFunc(theta_elec + 2*pi/3);

  // Back-EMF Calculation
  ea = 0.5 * Ke * w_mech * shape_a;
  eb = 0.5 * Ke * w_mech * shape_b;
  ec = 0.5 * Ke * w_mech * shape_c;

  // Neutral Point Voltage
  v_star = (v_a + v_b + v_c - (ea + eb + ec)) / 3.0;

  // Phase Voltage Balance
  v_a - v_star = R_phase * i_a + L_phase * der(i_a) + ea;
  v_b - v_star = R_phase * i_b + L_phase * der(i_b) + eb;
  v_c - v_star = R_phase * i_c + L_phase * der(i_c) + ec;

  // Torque Generation
  tau_em = 0.5 * Kt * (shape_a * i_a + shape_b * i_b + shape_c *
   i_c);
  tau_fric = Bv * w_mech + Coul * Modelica.Math.tanh(2.0 * w_mech)
   ;

  // Mechanical Flange Interface
  flange.tau = - (tau_em - tau_fric);

end BLDCMotor;
```

Listing 4.1: Modelica implementation of the BLDC Motor physics.

A test case was created and is reported in fig 4.11. To the current controller a reference current is given as input, the value of 17.82 can be found in the data-sheet

as the rated current I_{rated} [Amps Peak]. This value refers to a motor with sinusoidal back-emf and need to be adjusted for a trapezoidal shape by simple multiply by $\frac{\sqrt{3}}{2}$. The current controller receive the motor electrical angle θ_e and the values of the three-phase currents from sensors in the motor as input also. With these, it calculates the currents required for each phase, the voltages and then the duty cycles. These are used by the ideal inverter to provide the motor with the required phase voltage. Here, the supply voltage must be set at $\frac{V_{DC}}{2}$ since the voltage in each phase ranges from $-280V$ to $+280V$.

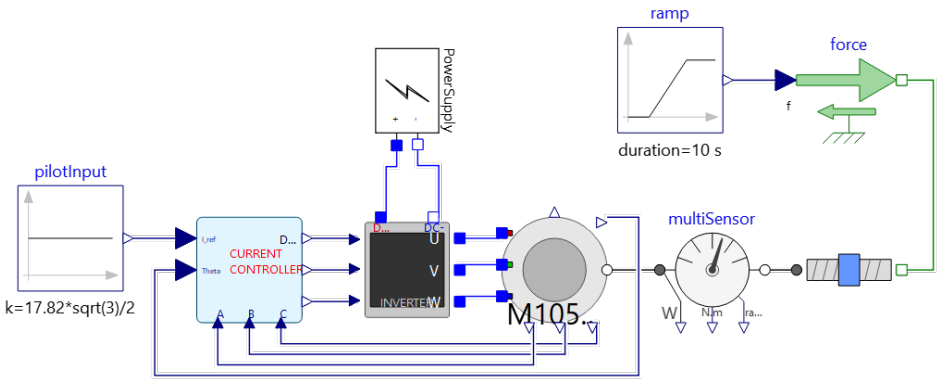

Figure 4.11: BLDC example test

As before, to asses the validity of the model the test case in fig 4.11 is used, results are shown in fig 4.12 and tab 4.6. As can be seen, the simulated performance are much closer to the data provided by Parker and this is believed to be sign of the good fidelity of the developed model. A thing must be said though, the behaviour of the motor when the speed is reversed is peculiar and unexpected, so it needs further investigation.

Table 4.6: BLDC motor characteristics

Parameter	Simulated Value	Expected value
Rated Power [kW]	3.72	3.739
Rated Torque [Nm]	7.11	7.14
Rated Current [A_{rms}]	15.4	15.43

Notes:
Rated values are measured @ 5000 rpm

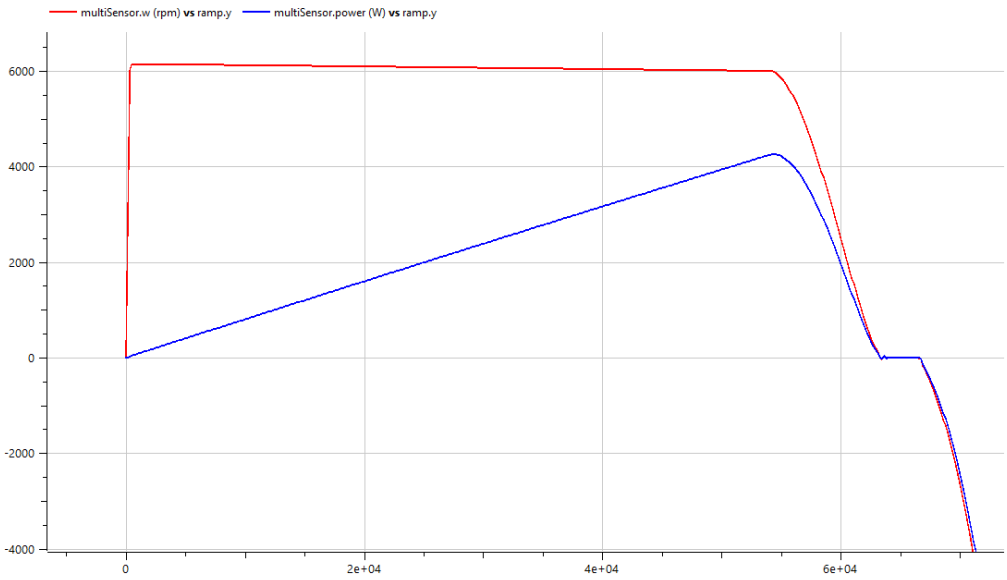

Figure 4.12: BLDC motor characteristics

4.3.3 Control Surface Geometric Model

The mechanical interface with the aircraft control surface was kept simple and modeled as an oscillating glyph. For many of the control surfaces this scheme can be quite a good approximation, especially for elevator, ailerons and rudder. For the flaps the kinematic is slightly more complex since the surface is not simply rotating but it also moves back and down. A simple scheme is reported in fig 4.13. The actuator is hinged to a the aircraft structure and its linear movement causes the rotation of the surface around another hinge. The most important parameter is the lever arm c because it dictates ratio between linear movement of the actuator and the rotation of the surface. The longer c is, the slower the surface moves, but also the bigger the maximum hinge moment becomes.

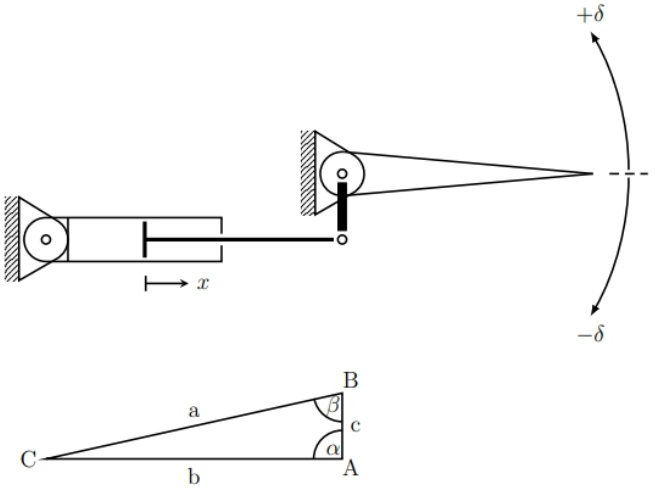

Figure 4.13: Control surface simple scheme

The equivalent model was developed looking to [30] as reference and is shown in fig 4.14. The geometric lengths are also reported in table 4.7 and come from [31].

Table 4.7: Geometric parameters

Parameter	Units	Aileron	Elevator	Rudder
a	m	0.2575	0.2575	0.2575
c	m	0.047	0.071	0.116

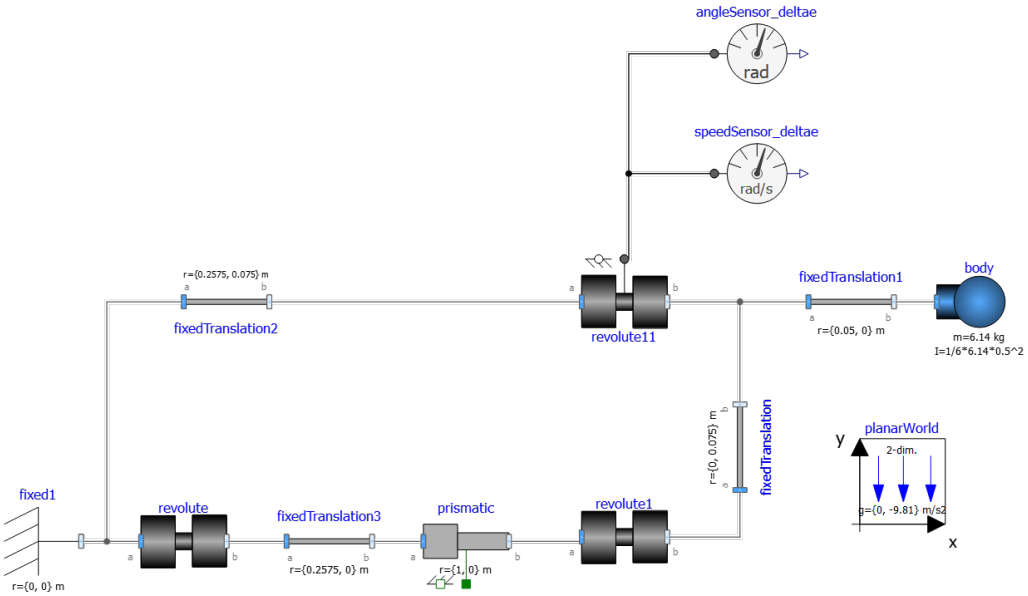

Figure 4.14: Elevator geometric model

4.3.4 Hydraulic Circuit Model

As already said at the beginning of this chapter, the hydraulic circuit has been modeled using the OpenHydraulics library. The architecture showed in fig 4.4 was then implemented in the model of fig 4.15. There is a fixed displacement pump that has an input flange for the torque and as output two ports, P and T. There are two relief-valves connected to the two line and they open once the pressure reaches a maximum threshold that can be set as parameter. There is an accumulator that feeds the two lines with two check-valves. The two lines are connected to the hydraulic cylinder chambers, which the size can be set by the user. In the model are present 4 components called VolumeClosed, they are used to avoid numerical problems during the simulation since the oil has a really high bulk modulus as they act as a leak for the oil. For the model to work it is necessary to insert the partial class *oil*, also the ambient parameters can be set using the *environment* class. All the components are connected using joints and no custom models were implemented.

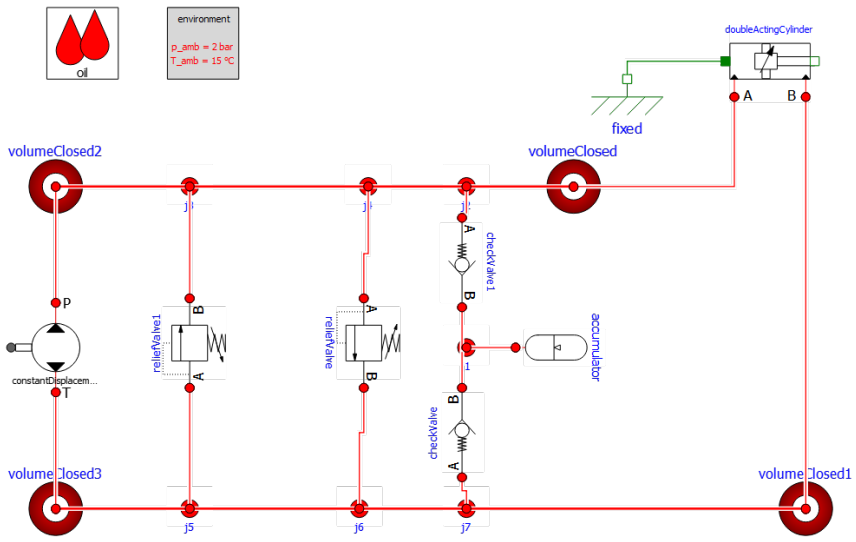

Figure 4.15: Hydraulic circuit model

The circuit was sized using [24] and [23] works since in both PhD thesis the reference aircraft was an A320. The value obtained where, however, slightly modified to have better results. The most touched values are the pump displacement V_{disp}, flow rate q_{nom} and maximum working pressure p_{max} since, with the values proposed in tab 4.8, simulations results were more satisfactory.

Table 4.8: Technical specifications for Hydraulic System Components

Parameter	Unit	Value
Fixed Displacement Pump		
Displacement (V_{disp})	m^3/rev	$1.5 \cdot e^{-6}$
Volumetric efficiency (η_p)	–	0.85
Dynamic Viscosity (Q_{th})	l/min	57.0
Bulk modulus (B)	N/mm^2	1020
Relief Valve		
Relief pressure (p_{set})	bar	290
Nominal flow (q_{nom})	m^3/s	0.01
Check Valve		
Open pressure (dp_{open})	bar	0.06
Nominal flow (q_{nom})	m^3/s	0.01
Accumulator		
Nominal liquid Volume (V_0)	m^3	0.011
Pre-charge Pressure (p_0)	bar	1
Max Working Pressure (p_{des})	bar	300
Hydraulic Cylinder		
Piston Diameter (D_p)	mm	50
Rod Diameter (D_r)	mm	0
Stroke Length (L)	mm	200
Piston Area (A_{piston})	cm^2	19.6
Piston mass (M_{piston})	Kg	1
Piston damping (B_{piston})	N·s/m	$1 \cdot e^4$
Nominal flow rate (q_{nom})	m^3/s	$1 \cdot e^{-4}$
Max pressure (p_{max})	bar	300

4.3.5 Actuator Full Models

All the proposed models can now be assembled to build the complete actuator. In order to test the models and assess their dynamic response, a simple PI position controller was implemented to make the actuator follow a position command. This configuration may not always be representative of actual operation, since the type of control applied depends on the pilot's input and on the operating mode of the Flight Control Computer; for instance, it may be necessary to follow a velocity or rate command instead of a position command. In such cases, the PI controller would need to be modified accordingly; however, the position-tracking scenario is assumed here as the most representative case for the purposes of this analysis.

The PI controller requires as inputs the commanded position, the measured position, and the measured velocity, and provides as output a reference current, which is subsequently fed to the current controller. All sensors within the model are assumed to be ideal in order to limit the overall complexity; nevertheless, the introduction of noise and filtering effects remains possible through dedicated blocks available in the Modelica Standard Library.

For all the proposed models, the dynamic response is commonly evaluated by applying a sinusoidal input command and observing the actuator's ability to track it. This is not the only viable approach, as other standard test inputs include the step, the ramp, the doublet, and the sawtooth signals. Additionally, an external load can be applied to the control surface by introducing a step torque directly at the surface hinge.

EMA Model

The model of the electromechanical actuator is fairly simple because, among the three architectures of fig. 4.1, the direct drive was chosen for being more compact than the geared one. In model terms, the EM is simply connected to the prismatic joint of the control surface via a screw. However, some time needs to be spent on this component.

In the Modelica Standard Library there is a component called IdealGearR2T that converts rotational motion into translation. This component is impractical since it requires the transmission ratio between the two flanges in rad/m, while in manuals the ratio is usually given as the pitch of the screw. To avoid having to convert the parameter every time, and being a simple component, a custom model was created that also takes stiffness into account. It is also easy to add friction forces, if necessary.

The final model is presented in fig 4.16.

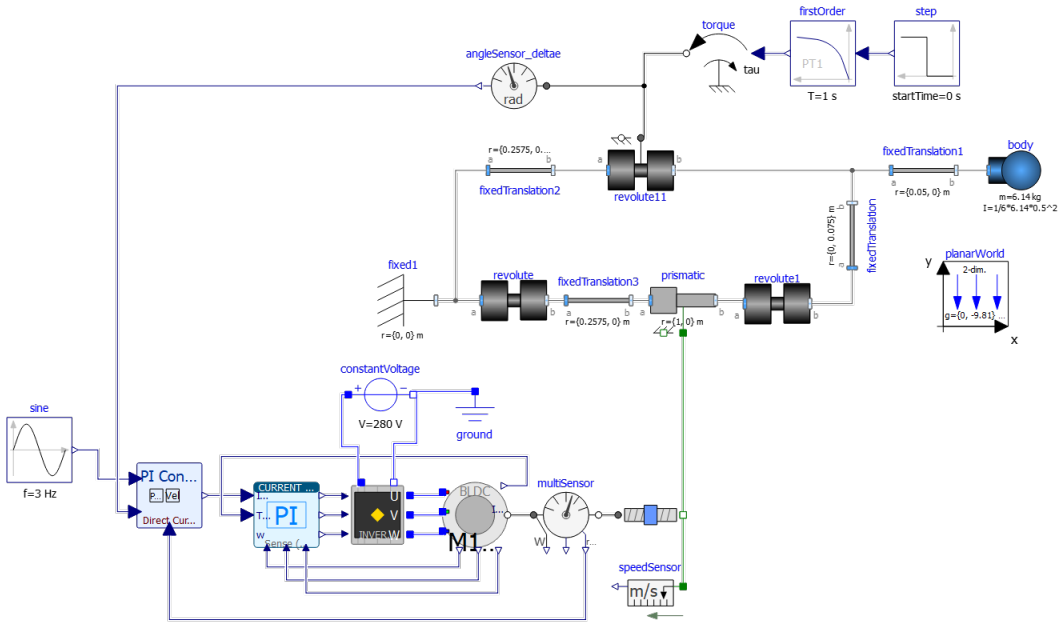

Figure 4.16: EMA full model

EHA Model

The electro-hydraulic actuator model is perhaps the most interesting since it contains three different physical domains: electrical, hydraulic and mechanical. For this reason the model is considerably slower than the EMA and also have many more components and equations to solve. Basically the only difference is the actuation: the simple ball-screw of the EMA is now replaced with the hydraulic circuit. The final model is shown in fig 4.17.

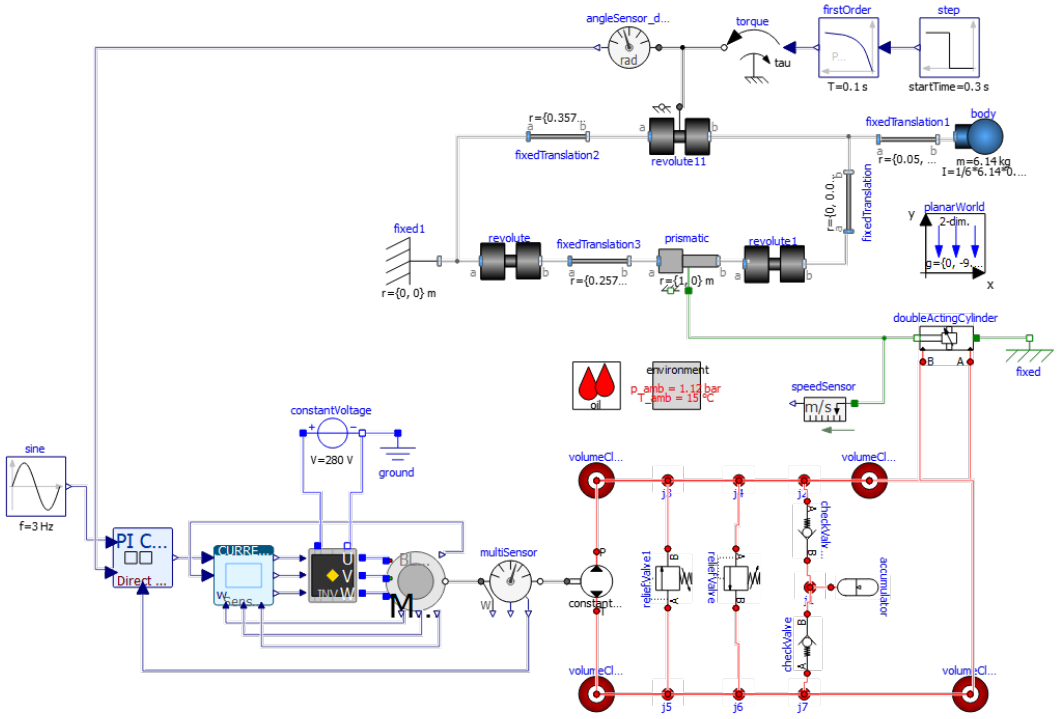

Figure 4.17: EHA full model

These models, as they were shown before, are not ready yet to be imported in modelica: the detail is too high and we cannot think of having the same complexity in SysMLv2, and it is not necessary at all. A model-of-models strategy was used: some components can be grouped together to create a sub-system. For example, there is no need to keep the complexity of the hydraulic lines if we want to analyze the full actuator behavior, so we can integrate everything into a higher-level model. The same goes for the kinematic chain of the elevator and for the electric motor and its power electronics. Even so, we decided to keep the electric motor and the electronics separated to see how this complexity reflects in SysMLv2. The final model can be seen in figure 4.18 and 4.19.

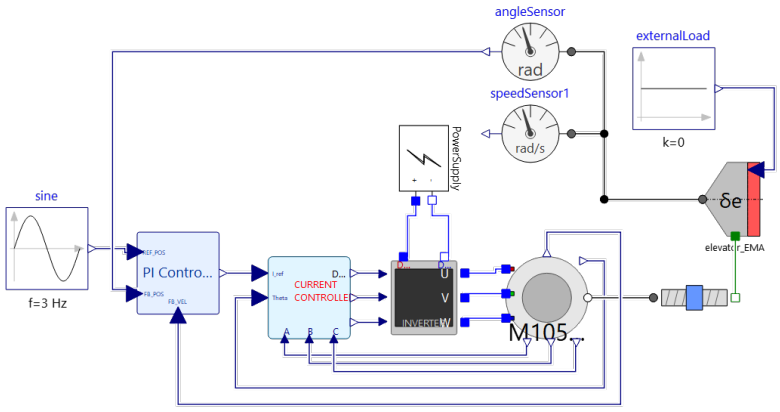

Figure 4.18: EMA with grouped components

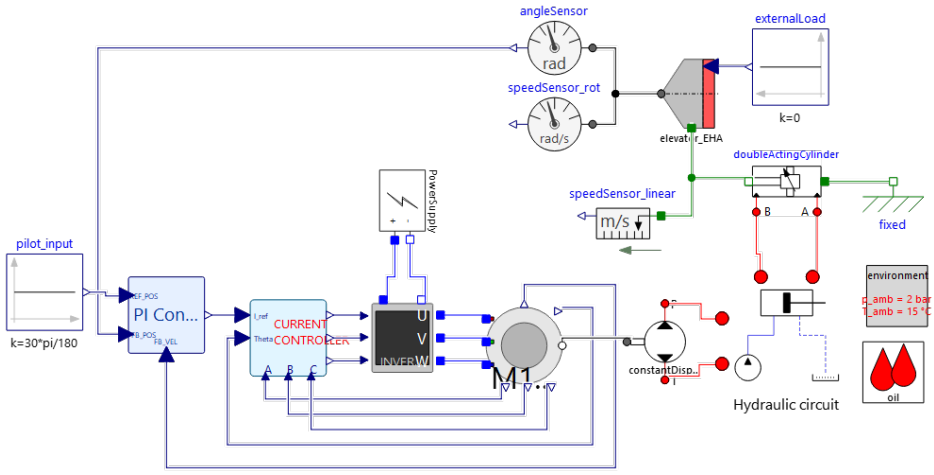

Figure 4.19: EHA with grouped components

Simulation Results

The behavior of the model was tested with different input commands and at zero external load. The first scenario is a command input of 5 deg at 0.5 Hz of frequency and the result is shown in fig 4.20. At this low frequency the behavior of the two model is basically the same, the two curves almost collide into one.

The second test is done with 5 deg but 3 Hz of frequency. The behaviour here is slightly different between the two actuators since the non-linearity are now manifesting. It can be seen in fig 4.21 that both actuator are now not fast enough to follow the sine wave and the typical saw-tooth shape can be observed, sign of a

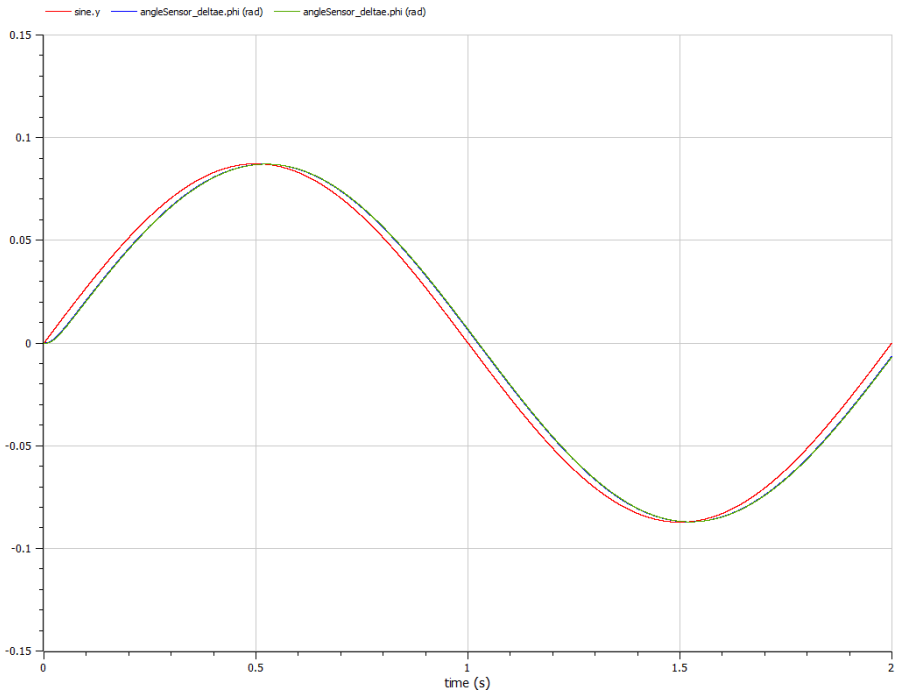

Figure 4.20: Actuator's response for a sine command of 5 deg at 0.5 Hz

system with saturation that limit its maximum velocity. The two curves here do not coincide perfectly because the two actuators are sized differently, in fact the EMA manifest a higher maximum speed than the EHA, sign that the pitch of the screw can be reduced a little. Since the sizing or optimization of parameters is not the scope of the thesis these differences were left and not investigated.

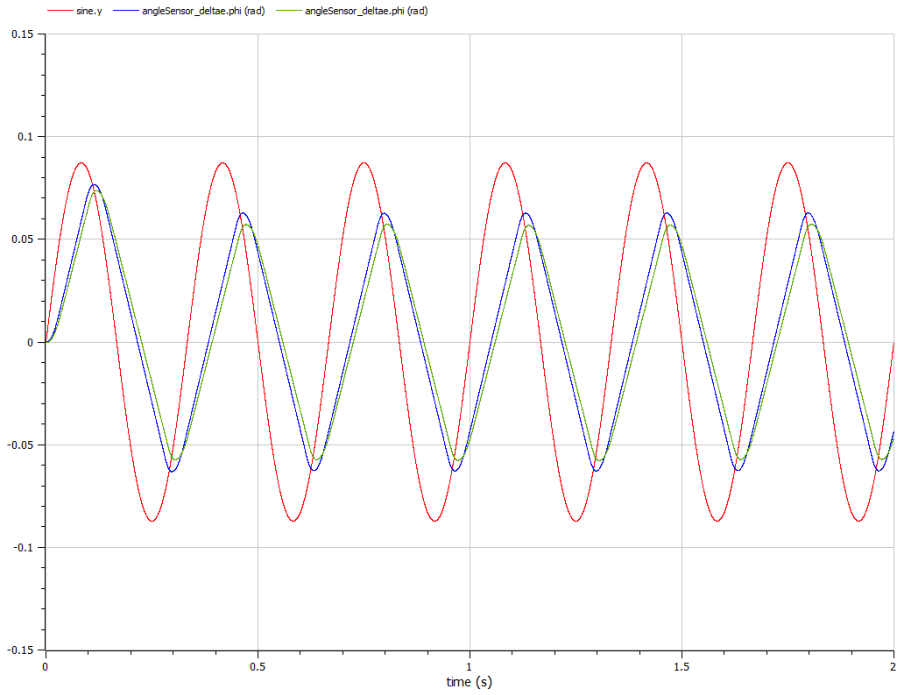

Figure 4.21: Actuator's response for a sine command of 5 deg at 3 Hz

Chapter 5

Requirement Virtual Verification

In this chapter, an example of the verification of a requirement for the Flight Control System will be demonstrated. SysMLv2 architecture models will be shown and explained (how to effectively create the simulation architecture test bench), following the proposed methodology, along with the outputs produced by the tool (dynamic models, .csv result files, etc.). It will then be shown how the requirement is validated and how its status is changed in SysMLv2.

Before proceeding, a brief explanation of how requirements can be verified in OpenModelica will be given. A thorough understanding of this is crucial, as it provides a clearer view of the potential of this virtual verification approach, while also highlighting its inherent limitations.

5.0.1 Validation of Static Requirements

Maximum no load velocity This requirement is simple to measure and assess, since it only requires placing a speed sensor in the model at the point where the measurement must be taken. For example, if the requirement concerns the linear speed of the hydraulic cylinder, a speed sensor can be connected to it. To assess the maximum speed there are multiple approaches, but the simplest one is to give a step command to the maximum deflection angle and read the speed from the sensor. An example is shown in fig 5.1.

Maximum hinge moment This requirement can be tricky to verify virtually. In this work, an external load linearly dependent on the deflection angle was applied. The external load is equal to the deflection angle, in degrees, multiplied by 1000. In this way, considering that the maximum deflection is 30 deg, the elevator

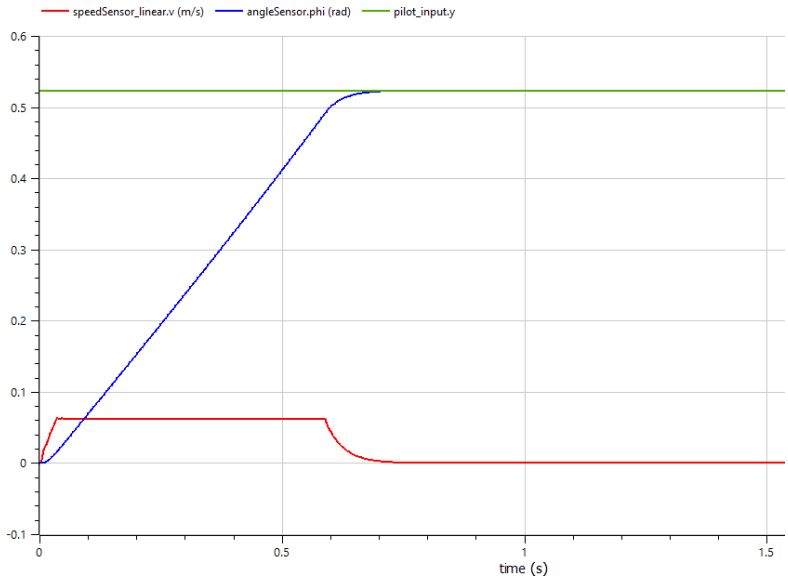

Figure 5.1: Maximum no-load speed

is expected to deflect by roughly 5 degrees before stalling. When running the simulation, the elevator stops moving once the maximum applicable load is reached, as can be seen in figure 5.2.

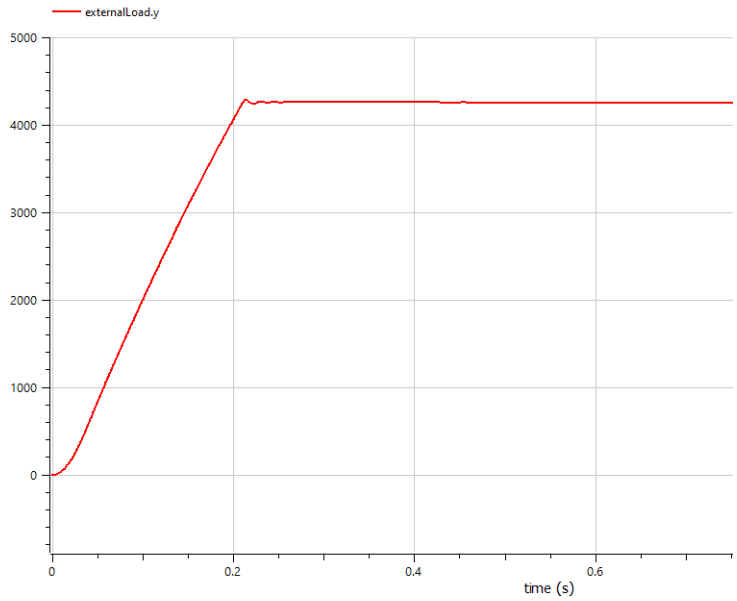

Figure 5.2: Maximum hinge moment

Steady state error This requirement can be easily assessed by giving a step input to the surface, preferably unitary, and measuring the output after a reasonable amount of time has passed. The difference between the actual position and the commanded one is the steady state error. The steady state error is theoretically zero if the controller has an integral part, otherwise, the bigger the external load the bigger the error.

5.0.2 Validation of Dynamic Requirements

One of the most important tools for frequency analysis is the Bode plot, from which most of the dynamic requirements can be discerned. The Python program developed for this purpose uses the *plotly* library to generate the plots on a web page; an example is shown in fig. 5.3. This plot shows the closed-loop transfer function $\theta_{actuated}/\theta_{commanded}$. From this plot it can be seen that, for very low frequencies up to 1 Hz, the response is almost perfectly in phase with the command, and the amplitude ratio is also close to 1. As the frequency increases, the actuator begins to damp the response and to lag behind the command. A small peak can be observed at around 100 Hz, where the phase also drops below $-180°$.

From the Bode plot we can also derive several important characteristics of the system, such as its bandwidth. Using the linearized system extracted with the *linearize* command in OMPython, we can calculate this bandwidth, which for the EMA and EHA developed usually lies between 3 and 5 Hz.

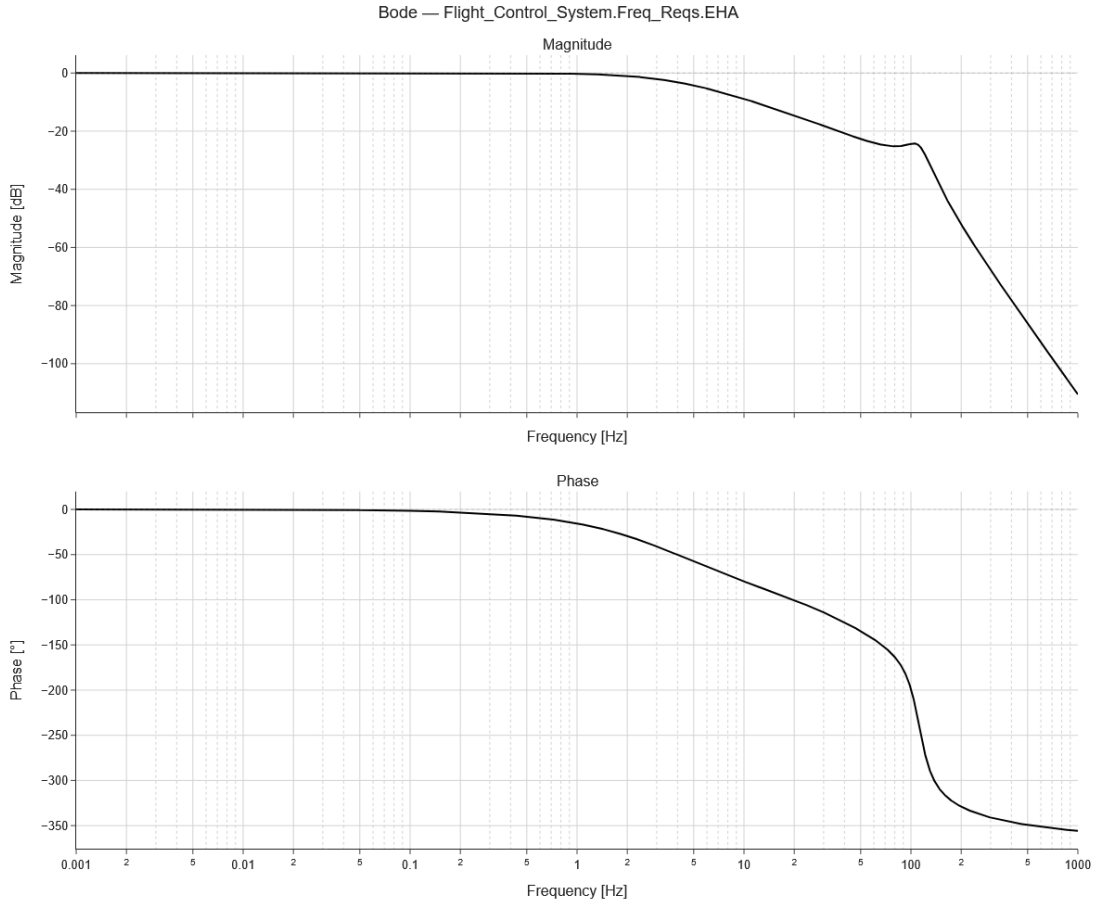

Figure 5.3: Bode Diagram for example EHA

5.1 Methodology Application

At this point, we have everything we need to define the requirement, derive an architecture, prepare a test bench, and verify it. We will now walk through the entire process using a concrete application case: the flight control system for the elevator. In particular, we will focus on the actuation system, with an electro-hydraulic actuator sized to match the characteristics of an Airbus A320.

As an example, we can take the maximum no load velocity being pretty straight forward to verify. We need to start by define the requirement in SysMLv2 exploiting the ODE4HERA library capability:

```
requirement <'R-001'> elevatorDeflectionRate : Requirement {
        doc
        /* Elevator shall deflect with rate 60 deg/sec at zero laod condition
    */
        subject Arch : FCSArchitecture;
        attribute DeflectionTest : TestEnvironment {
            :» ExternalLoad = 0 [SI::'Nm'];
        }
        require constraint { Arch.maxDeflectionRate >= 1 [SI::'rad/s'] }
        :» status = ReqStatus::Approved;
        :» version = "1.0";
        :» priority = ReqPriority::MustHave;
        :» implementationStatus = ReqImplementationStatus::InProgress;
        :» verificationStatus = VVStatus::NotStarted;
    }
```

Listing 5.1: SysMLv2 deflection velocity requirement

In 5.1 we identified the *FCSArchitecture* as subject of the requirement. The attribute *DeflectionTest*, of type *TestEnvironment*, defines the conditions under which the requirement must be satisfied — in this case, with an external load of 0 Nm. Than we set a constraint: we want the attribute *maxDeflectionRate* to be higher than 1 rad/sec (or equivalently 60 deg/sec). Then we can define some attribute like the priority and, most important, the verification status, here set as not started.

At this point, the high level system architecture should be defined. We start by defining the subject of the requirement, this can be used for multiple requirements as in 5.2 by creating multiple attributes. Then the constituting parts of the architecture are defined along with characteristic attributes. This parts are used in the design space while the values can be assigned in an architecture usage that is sub-setting the design space. Finally, we can use the architecture *satisfy* one or multiple requirements.

```
part def FCSArchitecture {
    attribute maxDeflectionRate : ISQ::AngularVelocityValue;
    attribute maximumActuatedLoad : ISQ::MomentOfForceValue;
        ...
    }

    // System Architecture Components
    part def ActuatorEHA {
        attribute MaxLinearSpeed : ISQ::SpeedValue;
        attribute MaxOutputForce : ISQ::ForceValue;
        attribute Mass : ISQ::MassValue;
        attribute maxPowerConsumption : ISQ::PowerValue;

        part motor : ElectricMotor;
        part pump : CircuitPump;
        part cylinder : HydraulicCylinder;
    }

    part def ElevatorSurface {
        attribute mass : ISQ::MassValue;
        attribute momentOfInertia : ISQ::MomentOfInertiaValue;
    }

    part def ElectricMotor {
        attribute mass : ISQ::MassValue;
        attribute Motor_type : ScalarValues::String = "Brushless DC Motor";
        attribute maxInputCurrent : ISQ::ElectricCurrentValue;
        attribute InputVoltage : ISQ::ElectricPotentialValue;
    }

    part def CircuitPump {
        attribute Pump_type : ScalarValues::String = "Fixed Displacement Pump";
        attribute PumpDisplacement : ISQ::VolumeValue;
        attribute maxOutputFlow : ISQ::VolumeFlowRateValue;
        attribute maxOutputPressure : ISQ::PressureValue;
        attribute mass : ISQ::MassValue;
    }

    part def HydraulicCylinder {
        attribute PistonArea : ISQ::AreaValue;
        attribute RodArea : ISQ::AreaValue;
        attribute MaxStroke : ISQ::LengthValue;
        attribute MaxPressure : ISQ::PressureValue;
        attribute nominalFlow : ISQ::VolumeFlowRateValue;
        attribute mass : ISQ::MassValue;
    }

    // System Architecture Design Space
```

```
#architectureDesignSpace part <fcs_ads> fcsArchitectureADS :
FCSArchitecture {
    doc /* Flight control system architecture design space */

    part actuator : ActuatorEHA;
    part elevator : ElevatorSurface;

}

#architecture part <fcs1> fcsElevatorHybridArchitecture subsets fcs_ads {
    part redefines actuator{
        redefines cylinder{
            :» PistonArea = 0.015 [SI::m^2];
            :» RodArea = 0.005 [SI::m^2];
            :» MaxStroke = 0.2 [SI::m];
            :» MaxPressure = 300e6 [SI::Pa];
                :» mass = 2.5 [SI::kg];
            }
        }
}

satisfy elevatorDeflectionRate by fcs1;
```

Listing 5.2: SysMLv2 FCS high level architecture

At this point, we can import the components from Modelica and use them to integrate the high-level architecture, obtaining the Model-Based Design solution. Figure 5.4 shows a comparison between the brushless motor developed in Modelica and its textual notation in SysMLv2. As can be seen, the complexity of the ports in Modelica is directly reflected in SysMLv2. This is an issue that must be addressed: developing well-structured models in Modelica translates to an easier design process in SysMLv2. For the case under analysis, the ports could have been grouped together.

After importing the components, we can proceed with defining the *mbd_architecture*. It is clear that this architecture is more complex than the previous one, as components such as the inverter or the controller were not included before. However, this level of complexity depends on the models imported from Modelica: for instance, we could choose to incorporate the power electronics directly into the BLDC motor model.

As shown in 5.3, the architecture references the previously defined high-level architecture. The same applies to every component, which links back to its corresponding part definition. For example, the attribute R was redefined to 0.8 to demonstrate that default values can be modified at this stage.

Finally, an issue with the connections becomes immediately apparent: without a graphical representation, managing a large number of connections purely in textual form can become overly complex and difficult to handle.

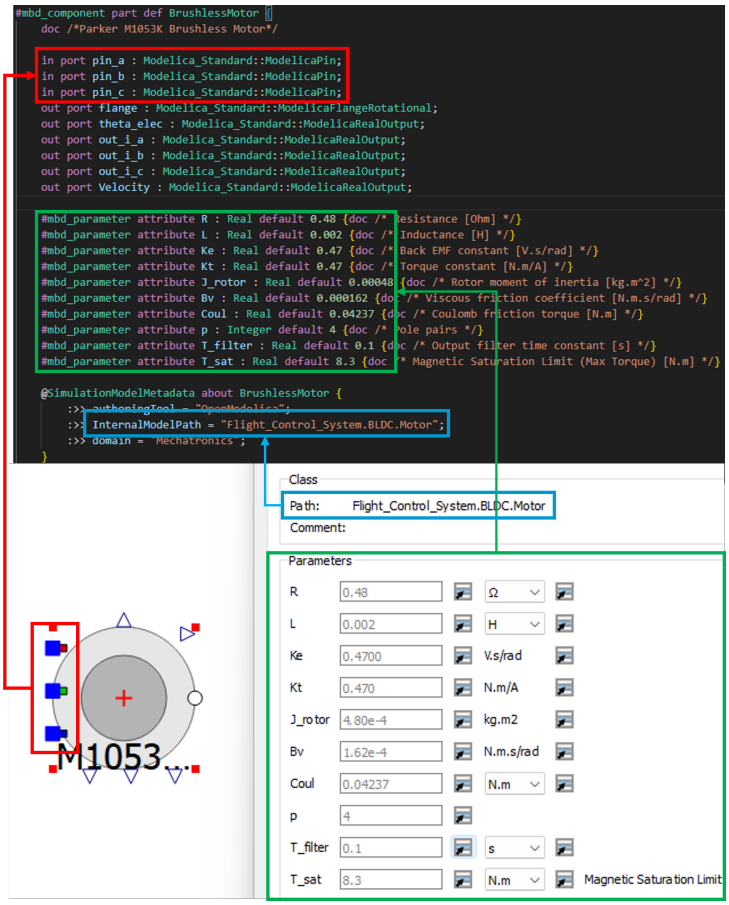

Figure 5.4: brushless motor: SysMLv2 and Modelica comparison

```
#mbd_architecture part FlightControlSystemArch1 {
        :» systemArchitecture references FCS_System_Architecture::fcs1;

        @SimulationModelMetadata {
            :» authoringTool = "OpenModelica";
        }

        #mbd_component part motor : Definitions::BrushlessMotor{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator.motor{
                :» R = 0.8;
            }
        }
        #mbd_component part inverter : Definitions::ThreePhase_Inverter_Average
    {
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator.motor;
        }
        #mbd_component part esc : Definitions::BLDC_Current_Controller{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator.motor;
        }
        #mbd_component part PID : Definitions::PI_Controller_Rotational{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator.motor;
        }
        #mbd_component part pump : Definitions::ConstantDisplacementPump{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator;
        }
        #mbd_component part cylinder : Definitions::HydraulicCilinder{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator;
        }
        #mbd_component part hydraulic_circuit : Definitions::Hydraulic_Circuit{
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    actuator;
        }
        #mbd_component part elevator_surface : Definitions::Elevator_EHA{
```

```
            :» systemArchitecture references FCS_System_Architecture::fcs1.
    elevator;
        }
        #mbd_component part power_supply : Definitions::PowerSupply{
            :» systemArchitecture references FCS_System_Architecture::fcs1;
        }
        #mbd_component part ground : ModelicaFixedTranslation{
            :» systemArchitecture references FCS_System_Architecture::fcs1;
        }
        #mbd_component part angle_sensor : AngleSensor{
            :» systemArchitecture references FCS_System_Architecture::fcs1;
        }

        // Connections
        connect PID.out_I to esc.I_ref;
        connect esc.d_a to inverter.d_a;
        connect esc.d_b to inverter.d_b;
        connect esc.d_c to inverter.d_c;
        connect power_supply.pin_p to inverter.dc_pos;
        connect power_supply.pin_n to inverter.dc_neg;
        connect inverter.phase_a to motor.pin_a;
        connect inverter.phase_b to motor.pin_b;
        connect inverter.phase_c to motor.pin_c;
        connect motor.theta_elec to esc.fb_theta;
        connect motor.flange to pump.flange_a;
        connect motor.Velocity to PID.fb_vel;
        connect motor.out_i_a to esc.I_meas_a;
        connect motor.out_i_b to esc.I_meas_b;
        connect motor.out_i_c to esc.I_meas_c;
        connect pump.portP to hydraulic_circuit.port_p;
        connect pump.portT to hydraulic_circuit.port_t;
        connect hydraulic_circuit.port_a to cylinder.port_a;
        connect hydraulic_circuit.port_b to cylinder.port_b;
        connect cylinder.flange_b to elevator_surface.actuator_lever;
        connect cylinder.flange_a to ground.flange;
        connect elevator_surface.rotational_hinge to angle_sensor.flange;
        connect angle_sensor.phi to PID.fb_pos;
    }
```

Listing 5.3: SysMLv2 FCS MBD architecture

Now it is possible to set the simulation to verify the requirement. We start creating a new architecture that contains a usage of the MBD architecture. Now we must define the boundary conditions, input/outputs and some high-level parameters. To add them we need to think how we want the test to be performed: for the deflection rate we want to measure the max rotating speed without ant external load applying to the elevator surface. There are many possibilities to do this, one was identified in the example 5.4: we give to the actuator the maximum deflection

command (30 degree) and we use a speed sensor connected to the rotational hinge of the elevator to measure the velocity. It is very important to define correctly the simulation settings, especially the stop time: we must give enough time to the actuator to reach the desired position. In this case, since the desired deflection rate of 60 deg/sec and the commanded deflection is 30 deg, we can end the simulation after 1 second. The desired output format is *.csv* and we only want to export the velocity measured by the speed sensor, we do so with *variableFilter*. Another important thing is the type of simulation: this requirement doesn't need a frequency analysis or to be linearized, so the attribute *SimulationType* was set to *"Time"*.

```
#mbd_architecture part Sim_ElevatorDeflectionRate {
    doc /* Test bench for R-001 - Elevator Deflection */
    @SimulationModelMetadata {
        :» authoringTool = "OpenModelica";
        :» SimulationType = "Time";
        :» VerificationLogic = VVLogic::Max;
    }
    #mbd_architecture part <fcs_model> FlightControlSystemModel1 :> FCS_MBD::
    FlightControlSystemArch1;
    #mbd_component part pilotInput : Modelica_Standard::ModelicaConstant{ :» k
    = 30*3.14/180; }
    #mbd_component part Constant_input_Torque : Modelica_Standard::
    ModelicaConstant{ :» k = 0; }
    #mbd_component part environment : Definitions::Environment;
    #mbd_component part oil : Definitions::OilTameplate;
    #mbd_component part speed_sensor : Modelica_Standard::RotationalSpeedSensor
    ;

    connect pilotInput.y to fcs_model.PID.ref_pos;
    connect fcs_model.elevator_surface.external_load to Constant_input_Torque.y
    ;
    connect fcs_model.elevator_surface.rotational_hinge to speed_sensor.flange;

    satisfy elevatorDeflectionRate;

    @SimulationSettings {
        :» startTime = 0.0;
        :» stopTime = 1.0;
        :» numberOfIntervals = 1000;
        :» tolerance = 1e-6;
        :» method = "dassl";
        :» outputFormat = "csv";
        :» variableFilter = "speed_sensor.w";
    }
}
```

Listing 5.4: SysMLv2 FCS simulation architecture

Once all the SysMLv2 models are ready, the tool can be executed. The initial outputs are the JSON files for the requirements and the simulation architecture, which are omitted here due to their length. Following this, the Modelica generator automatically builds the dynamic model in .mo format, as illustrated in Fig. 5.5. From the figure, it is possible to see that the components are instantiated in the graphical environment, but they lack an optimized layout. Due to time constraints, the current implementation simply places every component at the origin. Nevertheless, the underlying capability to auto-route connections and organize components logically remains available, utilizing the same approach used to generate the SysMLv2 graphical representation from textual notation.

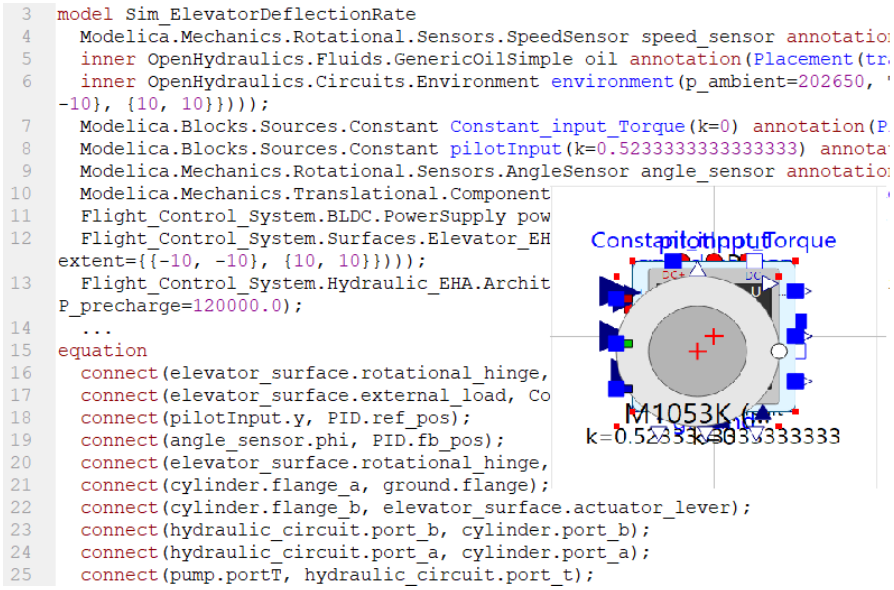

Figure 5.5: Generated model in textual and graphical notation

The model is then executed by the tool via OMPython that, since "Time" was specified in the simulation architecture, will simply launch the simulation with the settings defined before. The output of the simulation is a .csv file with the first column the values of the time, followed by all the variables specified in the attribute *variableFilter*, in our case only the velocity of the speed sensor.

Now the only remaining thing to do is verify the requirement. To do so the tool read the csv file with the results and follow the logic specified with the enumeration definition *VVLogic* defined in the simulation architecture. Here, the maximum value among the *speed_sensor.w* is taken. If this value is greater or equal to the threshold value in the requirement constraint, the test is passed and the requirement attribute *verificationStatus* is upgraded to *CompletedPassed*, otherwise to *CompletedFailed*.

Chapter 6

Conclusions

This thesis investigated the potential of SysML v2 for the automatic virtual verification of aeronautical system requirements, in the context of the ODE4HERA project. The execution of dynamic simulations is carried out in Modelica language since it can be freely used in OpenModelica environment, an open source software distributed by the Open Source Modelica Consortium (OSMC). While the integration of SysML and Modelica has already been investigated in the literature, existing approaches require manual intervention on the generated models, preventing the full automation of the process. Furthermore, all existing works targets SysML v1, leaving a technological gap with the new release SysMLv2. The research question that motivated this work was:

"How can we streamline and automate the generation and execution of dynamic simulations directly from system architecture models in order to verify system requirements?"

The proposed methodology demonstrates that it is possible to streamline and automate the generation and execution of dynamic simulation directly from SysML v2 architecture models, enabling the automatic verification of system requirements without manual intervention. Therefore, the research question can be considered answered, and achieved through the following:

- A proper mapping between system architectures in SysML v2 and the Modelica environment ensures consistency between the two domains.
- A Python transformation tool was developed to automatically generate simulation-ready Modelica models from SysML v2 architectural description.
- The requirements verification status is automatically updated in SysML v2 in order to close the loop between system design and virtual verification

- The decoupling between the Model-Based Design architecture, defined for all the requirements, and the simulation architecture, generated for a single requirement. This distinction allows the separation between design choices and the verification process since the simulation architecture can be adapted without affecting the system model.

The listed characteristics are what truly can enable a consistent automation of requirements verification through dynamic simulations.

The methodology has been validated by applying it to the Flight Control System for a More Electric Aircraft, based on the specifications of the Airbus A320. Dynamic models of Electro-Mechanical Actuators (EMAs) and Electro-Hydrostatic Actuators (EHAs) were developed and integrated in a model of the elevator control surface. These models are imported in SysML v2 and used to create Model-Based Design architectures, from which simulation architecture are derived for the verification of system requirements. Both static requirements — such as maximum no-load velocity, maximum hinge moment and steady-state error — and dynamic requirements — such as the system bandwidth — were successfully verified through the proposed automated workflow.

6.1 Limitations

It must be taken into consideration that the developed methodology and tool do not claim to be a final product. It was conceived as a proof-of-concept to asses the level of automation that can be achieved with the current state of the art. Many problems were faced during the development and some of them are still not solved.

One of the most significant limitations concern the management of ports and interfaces between SysML v2 and Modelica. When importing a component into SysML v2, the port names must match exactly the names in Modelica otherwise, the automatically generated dynamic model will not be able to resolve the connections. This is a strong limitation because it forces the system engineer working in SysML v2 to adapt to the naming convention of the Modelica component, requiring a minimum level of knowledge of the components before they can be effectively used. Also, it limits the the freedom of design choices. A decoupling of the port nomenclature between the two world would be a significant improvement, allowing the system engineer to work free of any constraints without being tied to the simulation models.

A related issue is the number of ports. Modelica components can be very detailed and have a great number of physical ports. In SysML v2, the system engineer is typically interested in representing logical and functional connections, and not all the physical interfaces. This forces the system engineer to already have

a deep knowledge of the low-level architecture at the SysML v2 level, which is often not the case in early design phases where the detailed architecture may still be undefined. A possible solution would be to wrap most of these ports into a single "black box" connection, exposing only the ports of interest at the architecture level and hiding the rest from the system engineer.

6.2 Future Steps

Looking at future developments, several directions can be explored to extend and improve the proposed framework.

The natural future step is the development of the inverse transformation, a Modelica-to-SysML v2 mapping. In the current methodology, the workflow is unidirectional: the system architecture is defined in SysML v2 and dynamic models are generated in Modelica. In practice, the models of the components are developed independently in Modelica by domain experts. A Modelica-to-SysML v2 transformator would allow these models to be automatically imported in SysML v2 and reflected in the architecture. More importantly, any modification made to the component in Modelica is automatically propagated to SysML v2. This would establish a bidirectional and consistent connection between the two environments, ensuring that the Model-Based Design architecture always reflects the latest state of the simulation models.

A second important development is the decoupling of ports between Modelica and SysML v2. As already discussed in the limitations, the current approach forces a tight coupling between the two environments. Implementing a port mapping strategy would allow the system engineer to work free of any bound with the low-level component, while the tool handles the mapping to the corresponding physical ports in Modelica independently.

Finally, the implementation of the proposed methodology with a borader Multidisciplinary Design Optimization (MDO) framework is crucial to unlock the potential of this work. In an MDO context, the system architecture is iteratively explored and optimized. The ability to automatically generate and execute dynamic simulations from architectural models makes the proposed methodology for this scope. This would enable architecture exploration, significantly reducing the manual effort required to evaluate each design candidate and accelerating the overall design process.

Bibliography

[1] National Aeronautics and Space Administration (NASA). *2.5 Cost Effectiveness Considerations*. https://www.nasa.gov/reference/2-5-cost-effectiveness-considerations/. July 2023 (cit. on p. 3).

[2] Alessandro Sguera and Giuseppe Di Crescenzio. «Modelling and Validating Product Line Variability for Nanosatellites based on CUBESAT Mission». In: Presented at the 7th International Workshop on Model-Based System Engineering (MBSE 2019). Feb. 2019. DOI: 10.13140/RG.2.2.14658.22722 (cit. on p. 3).

[3] D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin, and T. M. Shortell, eds. *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. 4th. John Wiley & Sons, Inc., 2015 (cit. on p. 4).

[4] International Civil Aviation Organization. *Trends in Emissions that affect Climate Change*. URL: https://www.icao.int/environmental-protection/trends-emissions-affect-climate-change (visited on 04/08/2026) (cit. on p. 5).

[5] Brian Pepper, Habibi Husain Arifin, Saulius Pavalkis, Jyothi Matam, and Ronald Kratzke. «Unlocking Synergy: Leveraging SysML and Modelica with Bi-Directional Transformation and Simulation Integration Standards». In: *INCOSE International Symposium* (July 2024). DOI: 10.1002/iis2.13239. URL: https://incose.onlinelibrary.wiley.com/doi/epdf/10.1002/iis2.13239 (cit. on pp. 7, 12, 14).

[6] Christian Nigischer, Sébastien Bougain, Rainer Riegler, Heinz Peter Stanek, and Manfred Grafinger. «Multi-domain simulation utilizing SysML: state of the art and future perspectives». In: *Procedia CIRP* 100 (2021), pp. 319–324. DOI: 10.1016/j.procir.2021.05.073 (cit. on pp. 9, 10).

[7] Dassault Systèmes and No Magic. *SysML Plugin Documentation. SysML Plugin 2024x*. No Magic Product Documentation. Sept. 2023. URL: https://docs.nomagic.com/spaces/SYSMLP2024x/pages/136724632/SysML+Plugin+Documentation (visited on 04/09/2026) (cit. on pp. 12, 13).

[8] Maryam I. Mukhtar and Bashir S. Galadanci. «Automatic Code Generation from UML Diagrams: The State-of-the-Art». In: *Science World Journal* 13.4 (2018), pp. 47–60. ISSN: 1597-6343 (cit. on p. 16).

[9] E. V. Sunitha and Philip Samuel. «Automatic Code Generation From UML State Chart Diagrams». In: *IEEE Access* 7 (2019), pp. 8591–8608. DOI: 10.1109/ACCESS.2018.2890791 (cit. on pp. 16, 17).

[10] Christian Buckl, Matthias Regensburger, Alois Knoll, and Gerhard Schrott. «Models for automatic generation of safety-critical real-time systems». In: *Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE Computer Society, 2007. DOI: 10.1109/ARES.2007.106 (cit. on p. 18).

[11] The Official OMG SysML Site. *OMG Systems Modeling Language*. Web. 2014. URL: http://www.omgsysml.org/ (cit. on p. 25).

[12] Kevin Hampson. «Technical Evaluation of the Systems Modeling Language (SysML)». In: *Procedia Computer Science* 44 (2015), pp. 403–412. DOI: 10.1016/j.procs.2015.03.054 (cit. on p. 25).

[13] Object Management Group. *Systems Modeling Language (SysML) Specification, Version 2.0*. Beta Specification. Object Management Group (OMG). 2023. URL: https://www.omg.org/spec/SysML/ (cit. on p. 27).

[14] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. «The Language of SysML v2 under the Magnifying Glass». In: *Journal of Object Technology* (2023) (cit. on p. 27).

[15] Starion Group. *What's new in SysML v2?* Web. Dec. 2025. URL: https://www.stariongroup.eu/whats-new-in-sysml-v2/ (cit. on p. 28).

[16] Vince Molnár et al. «Towards the Formal Verification of SysML v2 Models». In: *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. New York, NY, USA: ACM, 2024. DOI: 10.1145/3652620.3687820 (cit. on pp. 27, 29).

[17] Object Management Group. *SysML v2 Overview: Where We Are and How We Got Here*. Web. 2023. URL: https://www.omg.org/pdf/SysML-v2-Overview.pdf (cit. on p. 29).

[18] Sensmetry. *Syside Automator Documentation*. Sensmetry. 2026. URL: https://docs.sensmetry.com/automator/ (visited on 04/14/2026) (cit. on p. 42).

[19] URBANopt Contributors. *Modelica Builder*. 2026. URL: https://github.com/urbanopt/modelica-builder (visited on 04/14/2026) (cit. on p. 42).

[20] Open Source Modelica Consortium. *Scripting API — OpenModelica User's Guide*. Open Source Modelica Consortium (OSMC). 2026. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/scripting_api.html (visited on 04/17/2026) (cit. on p. 47).

[21] Guan Qiao, Geng Liu, Zhenghong Shi, Yawen Wang, Shangjun Ma, and Teik C. Lim. «A review of electromechanical actuators for More/All Electric aircraft systems». In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 232.22 (2018), pp. 4128–4151. DOI: 10.1177/0954406217749869 (cit. on p. 50).

[22] Matteo Dalla Vedova, Paolo Maggiore, Lorenzo Pace, and Simone Romeo. «Proposal of a model based fault identification neural technique for more-electric aircraft flight control EM actuators». In: *WSEAS Transactions on Systems* 15 (2016), pp. 19–27. ISSN: 2224-2678 (cit. on p. 51).

[23] Michael Anthony Cooper. «Simulating Actuator Energy Demands of an Aircraft in Flight». Accessed: 2026-01-12. PhD Thesis. Cranfield University, Feb. 2014. URL: https://dspace.lib.cranfield.ac.uk/handle/1826/8966 (cit. on pp. 52, 58–61, 63, 72).

[24] Tobias Röben. «Hybrid Actuation in Primary Flight Control Systems: A Force-Fight Inhibiting System Architecture». Analysis of active/active hybrid configurations (SHA/EHA/EMA) and force-fight mitigation strategies. Accessed: 2026-01-12. PhD Thesis. RWTH Aachen University, 2018. DOI: 10.18154/RWTH-2019-00621. URL: https://publications.rwth-aachen.de/record/753456/files/753456.pdf (cit. on pp. 52, 63, 72).

[25] Huatao Jin, Shuanglu Li, Yaobao Yin, Rui Guo, Cheng Fang, and Jiangkun Zou. «Enhanced Operation Mode Design and Motion Control of a Dual-Redundancy Electro-Hydrostatic Actuator». In: *Actuators* 13.12 (2024). DOI: 10.3390/act13120474 (cit. on pp. 52, 54).

[26] Chengwei Wang, Ip-Shing Fan, and Stephen King. «Failures Mapping for Aircraft Electrical Actuation System Health Management». In: *Proceedings of the 7th European Conference of the Prognostics and Health Management Society*. Turin, Italy: Prognostics and Health Management Society, July 2022, pp. 509–520. ISBN: 978-1-936263-36-3. DOI: 10.36001/phme.2022.v7i1.3354 (cit. on p. 53).

[27] Stefan Frischemeier. «Electrohydrostatic Actuators for Aircraft Primary Flight Control: Types, Modelling and Evaluation». In: *Proceedings of the International Conference on Recent Advances in Aerospace Actuation Systems and Components*. Available at TU Hamburg-Harburg, FST. Technical University Hamburg-Harburg, Section Aircraft Systems Engineering. Toulouse, France, 1997 (cit. on pp. 52, 56).

[28] Dominique Van den Bossche. «The A380 flight control electrohydrostatic actuators, achievements and lessons learnt». In: *25th International Congress of the Aeronautical Sciences*. ICAS. 2006 (cit. on p. 55).

[29] Dieter Scholz. «Development of a CAE-Tool for the Design of Flight Control and Hydraulic Systems». In: *Proceedings of the IMechE Conference on Recent Advances in Aerospace Hydraulics and Systems*. IMechE Conference Transactions 1996-8. Paper C505/9/011. Technical University Hamburg-Harburg. London, UK: Mechanical Engineering Publications Limited, 1996 (cit. on p. 57).

[30] Gianpietro Di Rito, Eugenio Denti, and Roberto Galatolo. «Object-Oriented Modelling of Flight Control Actuation Systems for Power Absorption Assessment». In: *Proceedings of the 27th International Congress of the Aeronautical Sciences (ICAS 2010)*. Comparison between SHA, EHA, and EMA technologies using Modelica. Accessed: 2026-01-12. Nice, France, 2010, pp. 1–11. URL: https://www.icas.org/icas_archive/ICAS2010/PAPERS/593.PDF (cit. on pp. 57, 61, 62, 70).

[31] Carlos Cabaleiro de la Hoz and Marco Fioriti. «New methodology for flight control system sizing and hinge moment estimation». In: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 236.12 (2022). Methodology for sizing EMA and estimation of hinge moments for primary control surfaces., pp. 2375–2390. DOI: 10.1177/09544100211063110. URL: https://doi.org/10.1177/09544100211063110 (cit. on pp. 58, 59, 70).

[32] Jan Roskam. *Airplane Design Part VI: Preliminary Calculation of Aerodynamic, Thrust and Power Characteristics*. Standard reference for hinge moment coefficient estimation and aerodynamic derivatives. Lawrence, Kansas, USA: DAR Corporation, 2000. ISBN: 978-1884885549 (cit. on p. 59).

[33] Stefán Baldursson. «BLDC Motor Modelling and Control: A Matlab®/Simulink® Implementation». Department of Energy and Environment. Master's Thesis. Göteborg, Sweden: Chalmers University of Technology, May 2005. URL: https://webfiles.portal.chalmers.se/et/MSc/BaldurssonStefanMSc.pdf (cit. on p. 63).

[34] Parker Hannifin Corporation. *Compumotor Catalog 2002: Positioning Control Systems and Drives*. Online; accessed 08-Jan-2026. Parker Hannifin Corporation. Rohnert Park, CA, 2002. URL: https://www.parkermotion.com/catalog_2002/catalog_2002.pdf (cit. on p. 63).