# POLITECNICO DI TORINO

*Master's Degree in Mechanical Engineering*


**Master's Degree Thesis**

**Integration of Model Based System Engineering (MBSE) environment with Multi-Disciplinary Optimization (MDO) approach applied to the Design of Automotive Electronic Hardware Architecture**

**Supervisors:**
Prof. Eugenio BRUSA
Eng.Massimo MANDORINO(Capgemini Italia)

**Candidate:**
Luca LOPEZ

Academic Year 2024–2025

## Abstract

In modern vehicles, the ongoing increase in capability, connectivity, and automation has led to a substantial rise in both the number and complexity of Electronic Control Units (ECUs). Balancing this growing complexity within strict cost, weight, and safety boundaries has become a major challenge for automotive manufacturers. Traditional document-based engineering approaches are no longer able to ensure consistency between requirements, design, and implementation, which motivates the use of more integrated and model-driven approaches capable of managing multidisciplinary trade-offs from the earliest phases of system development. The objective of this thesis is to develop and validate a framework for the preliminary design and optimization of automotive electronic architectures. In particular, the work addresses the question of how to allocate vehicle functions to ECUs in order to minimize cost and weight while ensuring compliance with performance and communication constraints. Furthermore, the framework aims to facilitate early phase decision making by supporting engineers in the exploration and identification of feasible designs across multiple architectural alternatives, even under complex design constraints. To achieve this goal, the thesis combines Model-Based Systems Engineering (MBSE) and Multidisciplinary Design Optimization (MDO). MBSE was implemented using Cameo Systems Modeler with the MagicGrid methodology to capture requirements, system functions, and architectures in SysML. The resulting model was then converted into structured CSV tables representing functions, computational loads, and communication signals. The optimization environment was developed in Python using the GEMSEO library, supported by a dedicated ECU database built on Intel and AMD FPGA/SoC devices. Two types of optimization problems were designed. In the single-objective case, both continuous and discrete formulations were applied: the continuous cost model was derived through curve fitting in MATLAB, incorporating computational performance and bus connectivity, while the discrete model relied on catalog-based ECU selection. In the multi-objective case, cost and weight were minimized simultaneously, producing Pareto fronts that describe trade-offs in the allocation process. In all formulations, constraints on computational load, bus capacity and maximum number of function for each ECU were enforced to guarantee the validity of the optimized architectures. The results of this thesis demonstrate that it is indeed possible to integrate MBSE and MDO environments, thereby improving both traceability across system representations and the quality of early decision-making in the design process. The optimized architectures obtained through the proposed framework show promising results in terms of cost reduction and mass efficiency, confirming the effectiveness of the approach. Nevertheless, future work should aim to extend the ECU database to provide greater variety and broader options for architectural allocation, as well as to introduce additional optimization disciplines such as power consumption, safety, and ECU placement. More broadly, the framework demonstrates the potential of combining system modeling and optimization to address the challenges of complex automotive electronics, supporting robust and informed decision-making from the earliest design stages.

# Contents

# List of Figures

# List of Tables

# Nomenclature

## Acronyms and abbreviations

| Acronym | Description |
|---|---|
| AAO | All At Once |
| AC | Air Conditioning |
| ALM | Application Lifecycle Management |
| BDD | Block Definition Diagram |
| CAN | Controller Area Network |
| CSV | Comma-Separated Values |
| DBSE | Document-Based Systems Engineering |
| DMIPS | Dhrystone Million Instructions Per Second |
| ECU | Electronic Control Unit |
| FMEA | Failure Mode and Effect Analysis |
| HMI | Human Machine Interface |
| HVAC | Heating, Ventilation, and Air Conditioning |
| IBD | Internal Block Diagram |
| IDF | Individual Discipline Feasible |
| INCOSE | International Council On System Engineering |
| LIN | Local Interconnect Network |
| MBPLE | Model-Based Product Line Engineering |
| MBSE | Model-Based Systems Engineering |
| MDA | Multidisciplinary Analysis |
| MDO | Multidisciplinary Design Optimization |
| MDF | Multidisciplinary Feasible |
| MoE | Measures of Effectiveness |
| OMG | Object Management Group |
| PDE | Partial Differential Equation |
| PDM | Product Data Management |
| SAND | Simultaneous Multidisciplinary Analysis and Design |
| SE | System Engineering |
| SysML | Systems Modeling Language |
| TLC | Telematic Locking Controller |
| UML | Unified Modeling Language |
| V&$V$ | Validation & Verification |
| XDSM | Extended Design Structure Matrix |

# Chapter 1

# Introduction

## 1.1 Context and Motivation

In the current engineering industry, especially in fields such as automotive, aerospace, and energy systems, the complexity of modern products has reached levels that traditional development methods are no longer well equipped to handle. Products are no longer standalone mechanical structures, but highly integrated systems that combine hardware, electronics, software, and communication networks, all of which must work together reliably and safely over time. This growing complexity has made it increasingly difficult to maintain consistency, traceability, and control throughout the design and development process. Continued reliance on document-based methods, combined with late-stage validation, often leads to errors, inconsistencies, and design misalignments, all of which can result in significant delays, rework, and increased development costs. These challenges become especially apparent in large-scale or multidisciplinary projects, where different teams handle specific parts of the system, often using their own tools, templates, and assumptions. Without a common structured model to guide development, key decisions can easily be made in isolation, without a clear understanding of how they impact the rest of the system. Requirements may be misinterpreted, changes in one subsystem may not be reflected in others, and the lack of automated checks makes it difficult to detect integration problems before they escalate.

As a result, many design flaws only become visible during the testing or integration phases when correcting them is much more costly and time consuming. This fragmented approach not only slows the development cycle, but also increases the risk of delivering products that do not meet performance, safety or compliance expectations. It highlights the urgent need for more structured and model-driven methodologies that can offer a unified view of the system from the earliest design stages.

In response to these challenges, many companies and institutions are increasingly shifting from traditional document-driven development methods to Model-Based Systems Engineering (MBSE). MBSE marks a fundamental shift in the way engineers define, communicate, and validate complex systems replacing fragmented document-based workflows with a model-centric approach in which information is formalized, structured, and consistently integrated from the earliest stages of development [13, 14].

MBSE aims to create a coherent digital representation of the system by integrating requirements, functional logic, architecture, behavior, and constraints into a single, consistent model. Unlike informal diagrams or static documentation, these models are developed using standard languages, most notably SysML, and managed with specialized tools such as Cameo Systems Modeler. The system model serves as a living artifact that evolves with the project and can be reused across phases and teams [15].
One of the key advantages of MBSE is its ability to ensure traceability across the system. Requirements can be directly linked to the functions that fulfill them, the components that implement those functions, and the interfaces that connect them. This allows teams to better assess the impact of changes, reduce ambiguities, and verify that every part of the system contributes to the intended behavior. Changes made in the model propagate automatically, improving consistency and reducing the risk of rework.
MBSE also enables early validation and verification (V&V). By simulating system behavior directly from the model or coupling it with domain-specific tools, engineers can identify design issues early in the lifecycle before costly implementation begins, reducing both technical risk and development time. [16]
In addition, MBSE supports cross-disciplinary collaboration, as engineers from different domains work on a shared, structured model. This helps prevent miscommunication and ensures that system-wide decisions are based on a consistent understanding. The reuse of architectural patterns and components across projects further improves efficiency and promotes standardization [17].
While MBSE is focused on form modeling and rationale of a complex system, it does not itself address how to make optimal design choices across many technical fields. MDO, Multidisciplinary Design Optimization, fills that void. MDO is an engineering discipline used for the solution of highly complex problems with many competing objectives and constraints in a broad range of disciplines, e.g. mechanics, electronics, control systems, and software.
The underlying assumption of MDO is that the system is addressed as a unit, rather than optimizing each subsystem in isolation; first developed in the aerospace sector to address the tightly coupled relationship of aircraft structure and flight, MDO has obtained additional relevance in automotive, energy, and robotics applications [18].
Originally developed to manage strongly coupled interactions among control, aerodynamics, and structure, MDO is now an essential tool to address complex multi-domain engineering problems. These days, however, MDO frameworks, often run in Python, MATLAB, or in-house systems, allow engineers to define objectives, place constraints, and automate searching for best design solutions across a wide range of parameters.
But in the vast majority of actual application, MDO remains used regardless of system architecture models. Optimization is typically performed against abstractions or solitary representations, never leveraging the high-fidelity structural and functional data already defined in MBSE models. Bridging the gap between the two approaches paves the way for model-driven optimization, in which architectural and performance decisions are made concurrently and constantly aligned with system-level requirements.

## 1.2 Thesis Objectives and Outline

As systems become ever more intricate, traditional design methods often rooted in isolated tools and discipline defined processes struggle to support the intricacies of interdependencies between architecture, performance, and system requirements. In response to these limitations, this thesis presents a novel solution, developed in collaboration between Politecnico di Torino and Capgemini, where I carried out my internship, that combines MBSE and MDO to enable improved decision-making during early design phases through increased traceability, system representation defined by purpose, and quantitative evaluation.
The objective of the research is to develop and validate a model-based workflow combining the description capabilities of MBSE and the analysis capabilities of MDO, with a key focus on creating a methodology to optimize the electrical and electronic architectures of systems. It aims to optimize an automotive Electronic Control Unit (ECU) architecture and its network, a product that embodies the need to balance functional, physical, and economic considerations using an integrated design approach.
The approach that is designed begins with the creation of a SysML system model within Cameo Systems Modeler, an MBSE software, where components, requirements, and functional relationships are clearly described and connected. The model serves as the structural and logical foundation used in the optimization process, which is implemented in Python, a programming language. Design parameters such as ECU assignment or network topology are examined with regard to two fundamental objectives: decreasing the overall system weight and minimizing the total cost.
Architecture and functional parameters are constrained by optimization immediately derived from the system model, such as ECUs count limits, redundancy policy, or communication dependency. By integrating tight MBSE and MDO layers, the proposed methodology enables a traceable and consistent design process in which architectural decisions are permanently evaluated against performance and cost requirements.
To verify its effectiveness, the approach is applied to a case study of the conceptual design of an automotive simplified ECU network. The optimization variables are defined as the number of functions and the outcomes are compared and contrasted for design quality and to determine the trade-offs between cost and weight, as well as the benefit of early and model-based optimization in guiding system-level design decisions. To guide the reader, the thesis is structured as follows:

- **Chapter 2** describes the methodology which is the foundation of the thesis. In this chapter, Systems Engineering is discussed first to show how it began to advance towards MBSE moved from a document-centric process to a model-centric process which supports consistency, traceability, and integration. Then the system life cycle and variety of V-model shapes are also introduced, followed by describing layered structures and abstraction levels that consider requirements that can provide input for design solutions. The chapter identifies SysML as the modeling language used in this thesis and points on its duality to express requirements, structures and behaviors into a combined representation. Next,MDO is introduced starting with the general 'all-at-once' formulation and works through the main finding monolithic

architectures, comparing their strengths and weaknesses. The last part of the chapter looks at the MBSE and MDO integration in order to show how MBSE models can ensure structured inputs to optimization, while MDO has quantitative evaluation tools to improve design decisions made by the architect. This integration is the methodological conceptual framework formation for the remainder of the thesis and provides the foundation for the case study in Chapter 3.

- **Chapter 3** describes the case study generated for the thesis, conducted in collaboration with Capgemini. The chapter introduces the case study with the project background and objectives, addressing the motivation for reducing cost and complexity in automotive electronic architectures. It then describes the modeling process employed in Cameo Systems Modeler, when stakeholder needs, context of system use, and use cases were formalized and refined into subsystem structures and exchange items. From activity diagrams structured tables are created to represent functional, computational loads and signal communication, which provide the linkage from the MBSE model to the optimization environment. After building a robust ECU database sourced from Intel and AMD, the work applies cost performance modeling via MATLAB curve-fitting, deriving continuous relationships between DMIPS, interface count, and cost. The optimization itself is then carried out under both single-objective and multi-objective formulations. In the single-objective experiments, continuous formulations allow for interpolated performance and cost surfaces, whereas the discrete case uses catalog-based ECU selection. The multi-objective work builds Pareto fronts illustrating trade-offs between cost, weight, and computational feasibility—all constrained by communication and performance requirements.

# Chapter 2

# Methodology

## 2.1 System Engineer main goals

The increasing complexity of modern technological systems has reshaped the role of engineers. Today, delivering a product that integrates multiple disciplines such as mechanics, electronics, software, and human interaction requires more than isolated technical expertise. It requires a figure capable of orchestrating the system as a whole. That figure is the System Engineer.
A System Engineer is a multidisciplinary integrator who ensures that all parts of a system function together as intended. This role spans from early-stage requirements elicitation to system design, integration, verification, and deployment.
The System Engineer is responsible for maintaining traceability between stakeholder needs and the final system architecture, managing trade-offs, and aligning technical decisions with business and operational goals.
At the core of this profession lies the concept of a system. According to ISO/IEC/IEEE 15288:2015, a system is: *"A combination of interacting elements organized to achieve one or more stated purposes."* [17]
This definition implies several key attributes:

- Components: A system is composed of multiple parts, which may be physical (e.g., hardware), logical (e.g., algorithms), or human (e.g., operators).
- Interactions: The components are not independent but interact through defined relationships functional, physical, or informational.
- Boundaries: Every system exists within an environment and has a boundary separating it from that context.
- Purpose: A system is not random; it is designed to fulfill a specific function or achieve a specific goal.

For instance, a modern vehicle can be considered a system that integrates propulsion, braking, user interfaces, connectivity modules, and environmental sensors. Each of these subsystems interacts to produce a coherent and safe driving experience. However, not

every collection of parts is a system. The components must be interdependent, and their configuration must give rise to behaviors or properties that cannot be attributed to individual elements alone a concept often referred to as emergent behavior.
As shown in the figure 2.1, when an automobile is considered as a system, it can be decomposed into multiple interrelated subsystems such as the transmission, exhaust, braking system, suspension, steering, electrical and electronic control systems, as well as safety and comfort features. Each subsystem performs specific functions, yet their operation is interdependent, collectively ensuring the vehicle's overall performance, safety, and user experience. This perspective highlights the multidisciplinary nature of modern vehicles, where mechanical, electrical, electronic, and software elements must be integrated into a coherent and reliable whole.

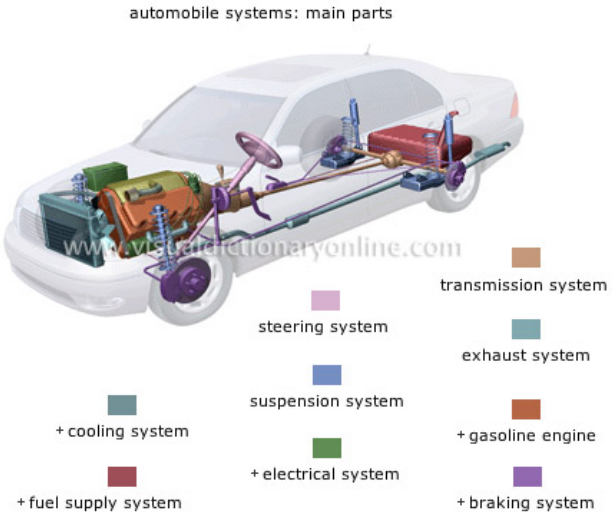


Figure 2.1: Example of system and sub-system in a vehicle [1]

Addressing this level of complexity requires a structured approach capable of managing the interdependencies between diverse components and disciplines. This is precisely the domain of Systems Engineering (SE), the discipline that enables the systematic development, integration, and validation of complex systems.
The International Council on Systems Engineering (INCOSE) defines Systems Engineering as *"a means to enable the realization of successful systems."* According to INCOSE, Systems Engineering *"focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem."* [19] Furthermore, *"it integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation."* Lastly, Systems Engineering *"considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs."* [19]
In alignment with this definition, Systems Engineering pursues a number of core goals

aimed at improving product development efficiency, technical performance, and lifecycle sustainability:

- **Manage increasing system complexity**: Modern products are no longer isolated components but integrated systems composed of mechanical, electrical, electronic, and software subsystems. As products incorporate more functionality and smart capabilities , the number of interfaces and dependencies grows exponentially. SE provides structured methods to manage this complexity by decomposing systems into manageable subsystems and modeling their interactions early in the development process. [2]

- **Ensuring end-to-end traceability across the lifecycle**: By adopting a model-based approach, SE enables the reuse of validated components and architectures, the automation of system documentation, and the alignment of system views across teams. Centralized digital repositories and configuration management tools allow for better knowledge sharing and consistency across development cycles and organizational boundaries. [2]

- **Improving safety, reliability, and diagnosability**: As system complexity increases, so do the risks associated with unexpected failures. SE tools and methods enhance safety by linking functional requirements to components and by enabling failure mode and effect analysis (FMEA), hazard tracking, and fault tree analysis. Additionally, system-level models support the development of diagnostic and prognostic capabilities that are essential for mission-critical systems. [2]

- **Integrating business and technical perspectives**: SE is uniquely positioned to bridge the gap between business goals and engineering execution. It ensures that systems are not only technically sound but also aligned with cost, schedule, and stakeholder expectations. This integrative function supports better risk management and promotes the delivery of quality systems that meet both user and enterprise needs. [2]

- **Reducing overall development costs and minimizing rework**: Systems Engineering advocates for a "left shift" in effort and decision-making investing heavily in early-stage activities such as requirement elicitation, trade-off analysis, and conceptual modeling shown in 2.2. By identifying design flow or integration issues early, SE helps avoid costly re-engineering in later phases, which is typically more resource-intensive and time-constrained. [2]

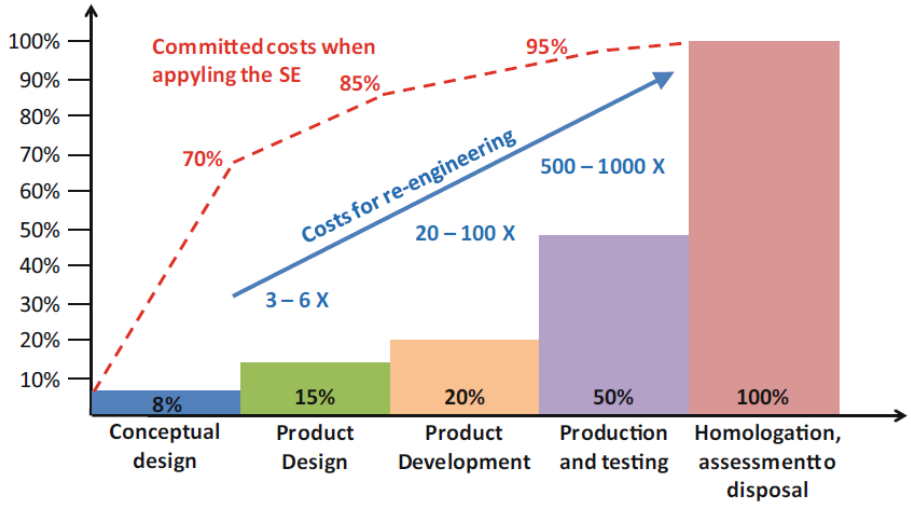


Figure 2.2: Comparison between the usual distributions of the percentage of total cost covered for the system development and that proposed by the SE [2]

In practice, Systems Engineering (SE) decomposes complex problems into manageable elements, examined from multiple perspectives to ensure thorough concept design and complete requirements elicitation. By integrating system models within a unified virtual environment, SE improves correlation between analyses, promotes a holistic system view, and provides a reusable rationale applicable across projects. [2]
In complex products with multiple subsystems, SE supports safety analysis by mapping clear links between requirements, functions, and components, enabling effective prediction of failure modes and ensuring full traceability. Functional modelling further aids in describing failures, while operational, functional, and architectural analyses enhance the design of effective monitoring systems and support detailed simulations.
SE bridges the gap between project management and product design, performing technical trade-offs, integrating multidisciplinary components, and maintaining alignment between requirements and the delivered product from concept through operation.

## 2.2 The evolution in Systems Engineering

In its earlier implementations, Systems Engineering (SE) was primarily conducted through a document-based approach. In this paradigm, system requirements, specifications, design descriptions, and verification procedures were produced, stored, and exchanged as static documents. These could take the form of requirement specifications, interface control documents, test plans, and design reports, often distributed across multiple formats and repositories. [19]
While effective in the early stages of SE adoption, this approach faced significant challenges as systems grew in complexity. Information was often fragmented and duplicated across different documents, creating inconsistencies and making it difficult to ensure that all stakeholders were working with the most up-to-date data. Updating a single requirement or design parameter could require manual revisions in multiple documents, increasing the likelihood of errors and omissions.
Moreover, document-based methods made traceability between requirements, design elements, and verification activities cumbersome. Identifying the impact of a design change on upstream requirements or downstream test procedures was often time-consuming and prone to oversight. This also limited the ability to conduct rapid trade-off analyses, as engineers needed to manually cross-reference disparate files rather than relying on a unified source of truth. [19]
As system architectures became more interconnected, particularly in domains such as aerospace, automotive, and defense, the document-based approach began to reveal additional shortcomings. Communication between multidisciplinary teams was hindered by the lack of an integrated environment, and the static nature of documents offered little support for real-time collaboration or automated consistency checks.
These limitations set the stage for the transition toward Model-Based Systems Engineering an approach that replaces static documentation with dynamic, interconnected system models. By representing requirements, architecture, behavior, and constraints within a unified modeling environment, MBSE enables continuous validation, automated traceability, and a holistic understanding of the system across its entire lifecycle. This shift not only addresses the shortcomings of document-based methods but also fundamentally transforms the way SE supports design, verification, and decision-making in complex engineering projects. [2]

The development of Systems Engineering (SE) can be described as an evolutionary process that has moved through four major stages, each driven by the increasing complexity of systems and the need for more integrated, accurate, and collaborative approaches.

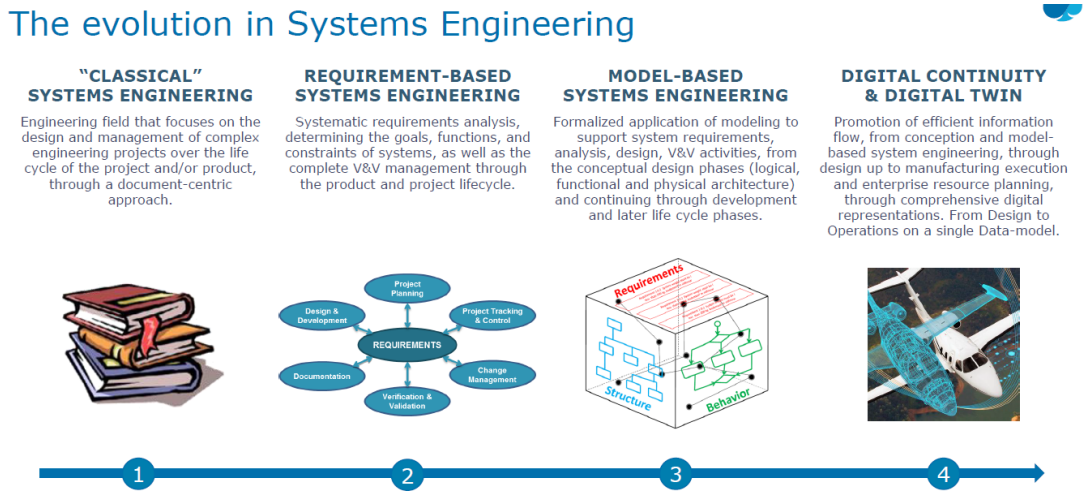


Figure 2.3: Evolution of System Engineering

- **Classical Systems Engineering** In its earliest form, SE was a document-centric discipline. The design and management of complex engineering projects were conducted over the life cycle of the system using static documentation as the main medium for capturing and exchanging information. Specifications, requirements, and design details were recorded in separate files or reports, often distributed across different teams and formats. While this approach provided structure and a degree of standardization, it suffered from significant limitations: fragmented information, time-consuming updates, and limited ability to maintain accurate traceability between requirements, design elements, and verification activities. [20]

- **Requirement-Based Systems Engineering** In this step SE was focused on placing requirements at the core of the engineering process. Systematic requirements analysis was introduced to clearly define system goals, functions, and constraints, ensuring that all subsequent design, verification, and validation activities were tied directly to these requirements. This approach improved visibility and control across the project lifecycle, supported change management, and provided a clearer link between requirements and the design and verification phases [20] . However, despite these advances, requirements were still mostly managed in document form, meaning that the process remained vulnerable to the inefficiencies of document-based workflows.

- **Model-Based Systems Engineering** To address the limitations of document-driven processes,MBSE emerged as a more dynamic and integrated methodology. MBSE formalizes the use of interconnected, digital models to capture requirements, architecture, behavior, and constraints in a single, authoritative environment. This

allows for real-time consistency checks, automated traceability, and more effective cross-disciplinary collaboration. By linking logical and physical architectures with functional analysis and verification data, MBSE enables engineers to understand the system holistically and evaluate trade-offs more efficiently [21]. This model-based approach also reduces the risk of inconsistencies, accelerates the design process, and enhances the ability to detect potential failures early.

- **Digital Continuity & Digital Twin** The most recent stage in the evolution of SE methodology is the integration of MBSE with the concepts of digital continuity and the digital twin. Here, the focus is on maintaining a seamless flow of information from concept to operations through a unified data model. Digital twins high-fidelity digital replicas of physical systems enable continuous monitoring, predictive maintenance, and performance optimization throughout the operational life of the system. This approach links design, manufacturing, and operational data, providing a closed feedback loop that supports faster decision-making, reduces lifecycle costs, and improves overall system reliability

## 2.3 MBSE: Introduction and benefits

As discussed in the last section, typical document-based systems engineering (DBSE) practices are clearly limited for complex and multidisciplinary activities. While document-based systems engineering is acceptable for simpler systems, the modern requirements of inter-disciplinary practice presents problems: limited integration between disciplines; poor management of consistency; the inability to maintain and trace relationships between requirements and design elements; and little opportunity for systematic re-use of knowledge. For these reasons, it is common for the technical baseline to degrade quickly across a project, with repercussions to cost, schedule, and quality.
To overcome these challenges, systems engineering has progressively evolved. A first step was the introduction of requirement-based systems engineering, which placed greater emphasis on the structured definition and verification of requirements across the life cycle. Building on this foundation, the discipline advanced towards MBSE, which today represents the standard paradigm for managing complexity. According to INCOSE, *MBSE is the formalized application of modeling to support system requirements, analysis, design, verification, and validation activities, beginning in the conceptual design phase and continuing throughout development and later life cycle phases.* [19].

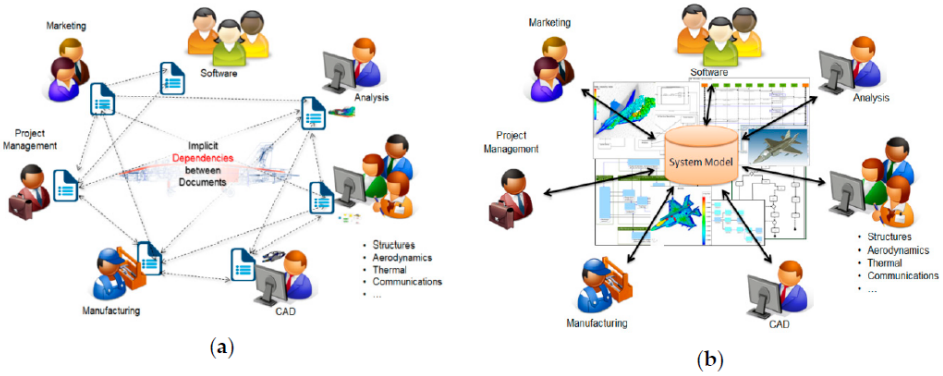


Figure 2.4: Comparison between traditional SE (a) and MBSE (b) [3]

The difference that MBSE provides, is in the replacement of documents with models as the primary engineering artifacts. Rather than static documentation, models are representations of facts and as such are authoritative and dynamic information sources that can show requirements, represent structures and behaviors, communicate to stakeholders, represent design trade-off analysis, and can stuff be executed and measure the system performance. In this respect, MBSE is not just a methodology, MBSE constitutes a change in the way we conceive of attributes like: requirements, architecture, behavior, and constraints into a coherent and evolving representations of the system. [22]
The motivations for adopting MBSE are deeply connected to the needs of contemporary industries. Among the principal benefits are:

- **Improve traceability**: In traditional document-based processes, the relationship between requirements, design elements, and validation steps is often fragmented

or lost across different documents.With MBSE, traceability is integrated into the model: each requirement can be traced to system functions, components, and associated test cases. This help engineer to understand how a decision propagates through the system, what requirement is being satisfied, and which planned validation procedure will confirm it.Such explicit traceability minimizes the risk of missing critical dependencies, facilitates impact analysis in case of design changes, and strengthens regulatory compliance in highly controlled industries.

- **Reusability** :Because system knowledge is stored in structured and machine-readable form, models can be adapted and refined for new projects, product variants, or successive generations of a product line.This continuity of knowledge prevents organizations from "reinventing the wheel" with every new program and fosters product line engineering and innovation by allowing organizations to leverage previously developed components.

- **Reduction of costs and time-to-market**: Traditional processes do not find design issues until late in the process when inconsistencies or requirement mismatches are discovered in prototypes or manufactured systems. MBSE lessens the problem, as it allows for verification and validation activities to be done much earlier in the life cycle utilizing simulation and model execution.Detecting problems at the conceptual or architectural stage prevents costly rework, limits delays, and contributes to faster delivery of robust products. Empirical studies confirm that early investments in MBSE reduce overall program costs, since errors corrected late in the process are exponentially more expensive. This reduction in cost can be see in 2.5 [3]

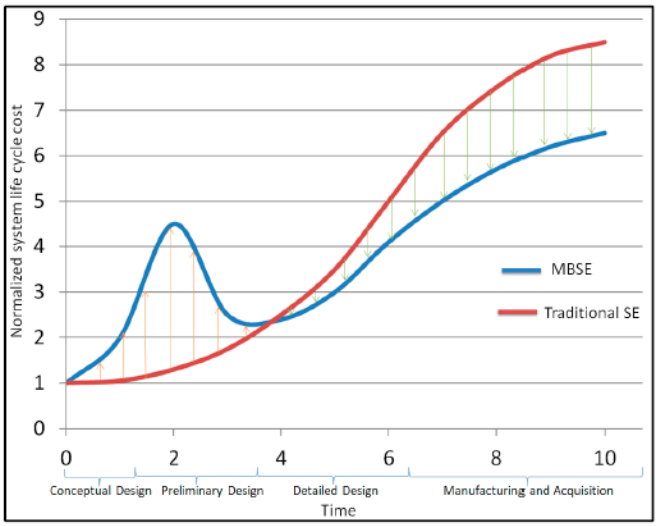


Figure 2.5: Comparison of costs between traditional SE and MBSE [3]

- **Improvement of communication**: Multidisciplinary teams often face challenges

in aligning terminology, understanding system behavior, or interpreting requirements. MBSE provides a shared, visual, and formalized representation of the system that acts as a *"common language"* across engineering domains. Diagrams, behavioral models, and structural views reduce ambiguities inherent in textual documentation, facilitating decision-making and consensus among engineers, managers, and external stakeholders. This makes achieving consensus or decision-making easier for the engineers, managers, or outside stakeholders and it is especially significant in international projects, where communication may involve linguistic or cultural barriers.

- **Risk management and quality assurance**: Executable models allow engineers to evaluate design alternatives, simulate system performance under different conditions, and test critical scenarios before physical implementation. This anticipatory capability makes it possible to identify risks early, assess their potential impact, and design mitigation strategies proactively. As a result, the quality of the final system improves, not only in terms of compliance with requirements but also in terms of robustness, safety, and resilience to failures.

Having outlined the main benefits of adopting Model-Based Systems Engineering, it is important to clarify how these advantages will be further developed in the following chapters. MBSE should not be regarded simply as a set of abstract principles; rather, it constitutes a methodological framework that reshapes the way system life cycles are conceived and managed. As emphasized in the INCOSE Systems Engineering Handbook, MBSE integrates seamlessly into established life cycle models, providing digital continuity that extends from the earliest phases of concept definition through verification, validation, operation, and eventual decommissioning. [19]
This continuity ensures that the links between requirements, design, and validation are not only preserved but dynamically updated as the system evolves.The next chapter will therefore explore how traditional system life cycle representations, such as the waterfall, the V-model, and the spiral model, can be reinterpreted in light of MBSE practices, highlighting their complementarities and limitations when applied to contemporary engineering contexts.
Closely related to this discussion is the question of abstraction layers and system levels, which will be addressed in the subsequent section. MBSE facilitates the definition of consistent system views across multiple levels of abstraction, ranging from stakeholder needs and high-level functional representations down to logical architectures and detailed physical implementations. By maintaining coherence between these levels, MBSE reduces the risk of inconsistencies and ensures that every design decision remains traceable back to its original requirement.
In conclusion, we will focus specifically on the significance of languages and tools that facilitate MBSE in practice. The application of formalized modeling languages, typically SysML, provides the expressiveness needed to represent the structural, behavioral, and parametric facets of a complex system in a shared semantic space. Together with dedicated tools offering model execution, requirements management, and automated analysis capabilities, SysML enables the methodological and operational benefits described above. In this way, MBSE can become not only a formalized conceptual change, but also

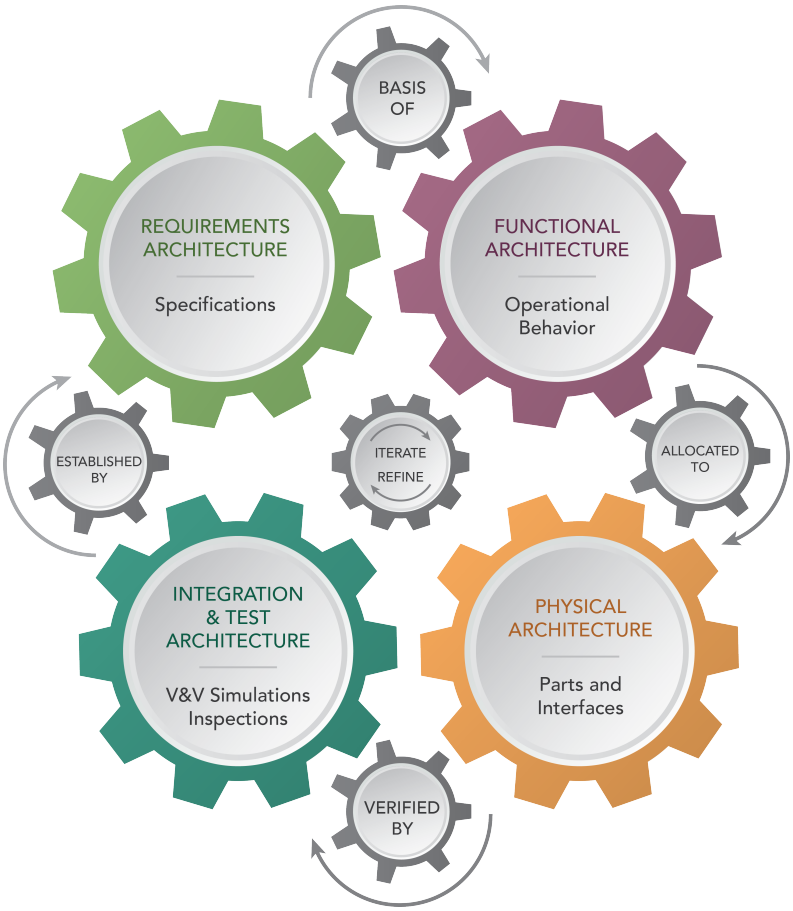


Figure 2.6: MBSE architecture [4]

an embodied practical approach based on tools and methods that enable the effective application of MBSE principles in an industrial setting.

## 2.4 The System Life Cycle and the V-Model Framework

In systems engineering, every product is understood not only as a physical artifact but as a system that evolves through a life cycle. The life cycle encompasses all stages of existence, from the initial conception of needs to final decommissioning and disposal. According to ISO/IEC 15288, a life cycle is defined as the *"evolution of a system from conception through retirement"*, which includes conception, development, production, operation, maintenance, and disposal [17].
Taking and applying a life cycle framework is critical for two reasons. First, it creates a full spectrum view of engineering behaviors, understanding that activities and decisions made during design and production have immediate impact on operation, sustainment, and retirement. Second, it underscores the thread of stakeholder needs; requirements established during conception must remain true and traceable during the use of the system, while feedback during operation can guide redesign and subsequent generations of products. Several models have been proposed to represent this life cycle graphically. Among the most common are the Waterfall model, the V-Diagram, and the Spiral model. Each of these emphasizes different aspects of the development process:

- **Waterfall Model** is one of the earliest and most intuitive representations of a system development life cycle. It visualizes the process as sequential steps flowing downward from requirements analysis to design, implementation, testing, deployment, and disposal [23]. Its appeal lies in its simplicity: each phase produces outputs that feed the next, offering a clear pathway for project planning and management. However, the model assumes requirements can be fully defined at the start and remain stable, which is rarely the case in complex systems. Since it lacks explicit feedback loops, errors discovered in later phases become costly to fix. For this reason, Waterfall is often considered too rigid for modern engineering, though it retains value in projects with stable requirements and low uncertainty.

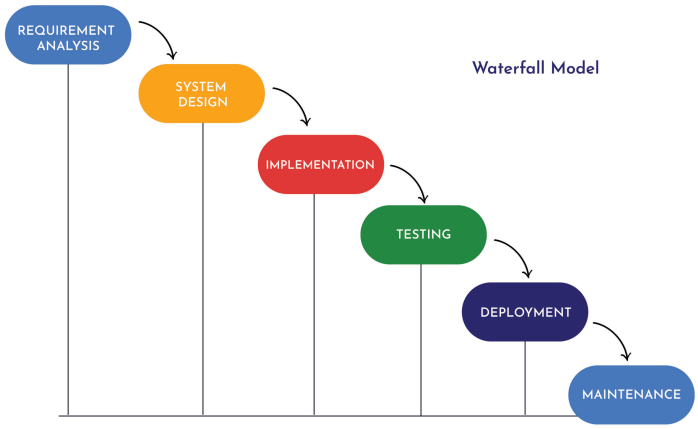


Figure 2.7: Model of the product lifecycle described by a waterfall diagramg

- **Spiral Model**,introduced by Boehm in 1988, extends the life cycle perspective by placing iteration and risk management at the center of development [24]. It was proposed to overcome the limitations of the Waterfall approach, emphasizing the recursive nature of Systems Engineering. Development proceeds step by step, circle by circle, with each iteration refining the previous one, particularly when requirements are involved. As the spiral advances toward the center, the product gradually reaches completion through requirement analysis, functional and physical modeling, and final validation. This model highlights the need for continuous reassessment, but some limitations remain. In safety-critical systems, for example, non-functional and dysfunctional analyses must be integrated early to ensure reliability and prevent severe failures. Moreover, effective product development should also consider business aspects from the outset, avoiding excessive costs or weak market positioning. For these reasons, the spiral lifecycle is often combined with defined processes and standards that continue to evolve in both industry and academia. [2]

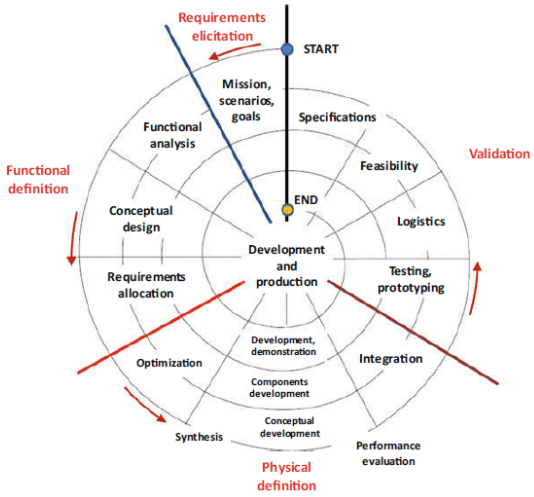


Figure 2.8: Sketch of spiral diagram applied to the product development in systems engineering [2]

- **V-Diagram**: To address the limitations of linear representations such as the Waterfall model, the V-Diagram has been widely adopted in systems engineering as a way to describe the fundamental stages of system development [2]. This diagram, shaped like the letter "V", illustrates the full trajectory of system engineering activities: the left branch descends from the identification of stakeholder needs to increasingly detailed design specifications, while the right branch ascends through integration, verification, validation, and final deployment (2.9. The path begins at the upper left corner with the early concept definition and progresses downwards into more detailed design, before rising again through testing and validation to the upper right corner, which represents deployment, service, and eventual disposal.

The holistic nature of the diagram is one of its strengths, as it makes explicit that system development spans the entire life cycle, from conception to retirement.The diagram portrays a top-down approach: engineers start with the system as a whole, and then decompose it down to subsystems, components, and parts. This perspective is different than in traditional material products, where each component is designed and manufactured separately, in isolation, and then assembled as a system.

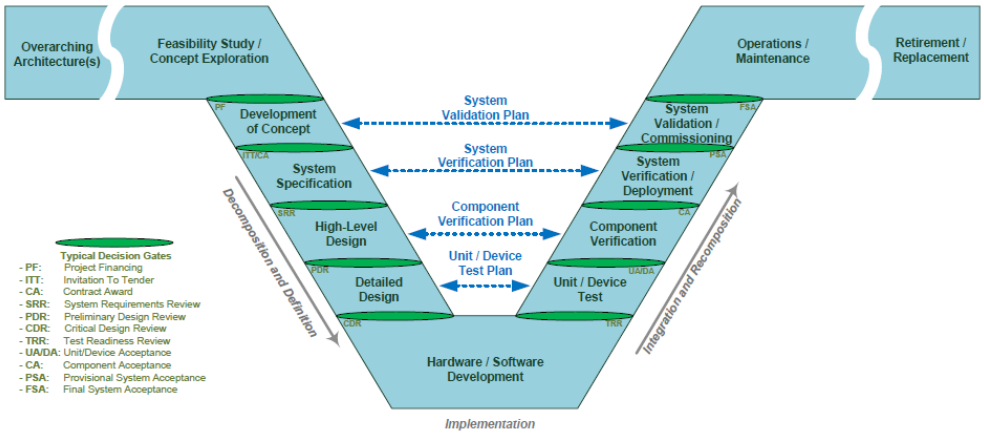


Figure 2.9: V-diagram and decision gates

The V-Diagram recognizes the need to be connected between the two perspectives; every design activity on the left must equal a verification or validation activity on the right. In this representation, the left-hand branch is often associated with Application Lifecycle Management (ALM), which involves requirements elicitation, functional decomposition, and architectural definition. The right-hand branch corresponds to Product Data Management (PDM), encompassing integration, assembly, verification, validation, and service. Importantly, ALM and PDM should not be seen as disconnected or sequential processes. Instead, the diagram emphasizes their interdependence: for every function defined during the design phase, a corresponding test must be foreseen to ensure that the requirement is properly verified and the underlying stakeholder need is satisfied. This direct symmetry is one of the reasons the V-Diagram has become a cornerstone of systems engineering. At each major transition of the V-Diagram, decision gates are established to formally assess progress and determine whether the project can advance to the next phase, ensuring alignment with stakeholder needs and system objectives [19]. The diagram also provides a chronological guideline for development.On the left-hand side, the diagram starts with identifying customer needs, and it is evident stakeholders drive requirements and specifications. By performing stakeholder identification, analysis, and mapping, we can ensure all relevant actors, including people, processes, systems, and devices, are on the radar in this early period of the process, typically through use cases. On the right side, each of these requirements is systematically validated, moving from component testing to subsystem integration and finally to

system-level acceptance.Additional considerations, such as transportation, installation, and disposal, are also placed within this right-hand branch, reflecting the full span of the system life cycle. Although the V-Diagram provides clarity, it is not without limitations.One criticism is that it shows requirements elicitation as a one-time event established at the beginning of a development process, but requirements in practice evolve throughout design loops and performance assessments. Therefore, revisions to the diagram have incorporated feedback loops (2.10) in the systems specification process to symbolize the iterative nature of requirements as shown in figure below.

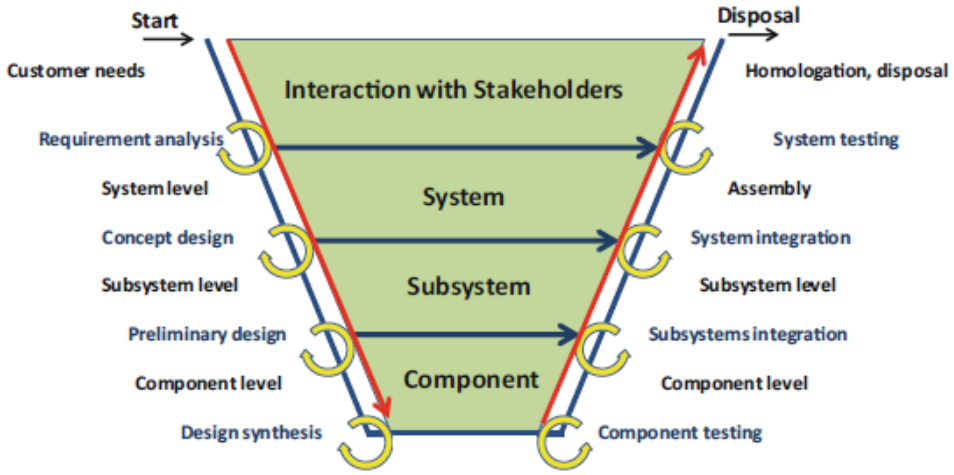


Figure 2.10: V-diagram applied to the product development in systems engineering [2]

Contemporary systems engineering requires a broader perspective that integrates monitoring, operator training, and continuous improvement, in line with Industry 4.0 paradigms. Moreover, the focus has expanded from individual systems to entire product lines, introducing Model-Based Product Line Engineering (MBPLE) as an extension of MBSE to capture variability and reuse across evolving products. [2] For these reasons, the V-Diagram should never be interpreted as a fixed process and should be considered a flexible framework that facilitates traceability from requirements to validation; and allows for iterative change during the evolution of the system. Also, when combined with Model-Based Systems Engineering, the V-Diagram could be interpreted as both a methodological framework and a practical roadmap for the development of complex and safety-critical systems.
The figure below illustrates an evolution of the traditional V-model through the integration of Model-Based Systems Engineering (MBSE). In the classical representation, the V-model is essentially a sequence where requirements are decomposed on the left side and then validated through integration and testing on the right side. Here, however, the model is enriched with a clear separation between the system level and the component level, highlighting how MBSE bridges the gap between stakeholder requirements and the detailed development of components.

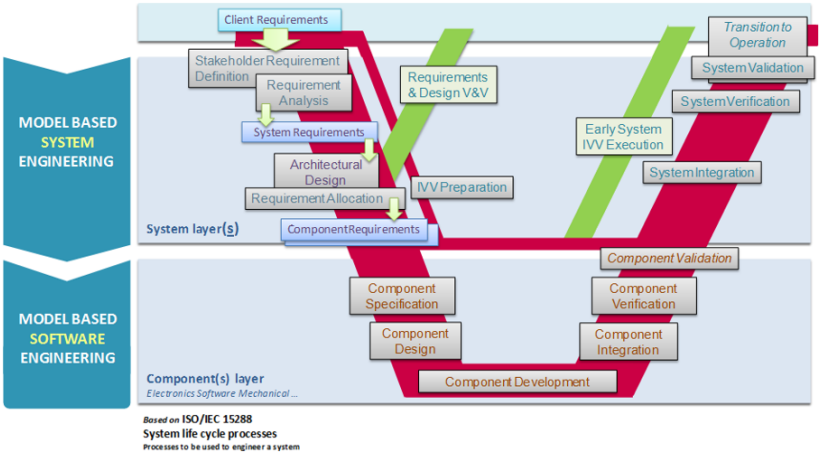


Figure 2.11: V-diagram applied to the product development in systems engineering

At the system layer, requirements are not simply listed as static documents but are formalized into models. This allows for architectural design, requirement allocation, and early verification and validation to be carried out in a continuous and traceable way. Moving down to the component layer, the process covers specification, design, development, and integration, showing how system requirements are translated into technical solutions.

What distinguishes this MBSE-based V-model from the basic one is the stronger emphasis on iteration and early validation. By enabling the execution of verification activities already in the design phase, it reduces the likelihood of overlooking dependencies and makes it easier to detect inconsistencies before costly rework is required. In this sense, MBSE ensures digital continuity throughout the lifecycle, from stakeholder needs all the way to system integration and operation, offering a more robust and interconnected framework than the classical V-model.

## 2.5 Layered Structures and Model Levels in MBSE

As previously discussed, one of the defining challenges of contemporary systems engineering lies in the increasing complexity of modern systems. These systems are no longer linear or isolated artifacts but rather interconnected entities that must satisfy different stakeholder needs while operating across dynamic environments throughout their entire life cycle. Such complexity cannot be effectively addressed if the system is analyzed from a single perspective or confined to one level of detail.
To address this issue, MBSE presents the two notions of layers and levels, which provide a structural underpinning for modeling and reasoning about complex systems. Layers are perspectives on the system: operational, functional, and physical views focus on separate questions and interests. Levels capture a hierarchical decomposition of the system, defining its global architecture, systems, subsystems, components, and lower-level parts. By combining these two dimensions, MBSE enables both vertical traceability (ensuring requirements flow consistently from system-level needs to detailed implementations) and horizontal consistency (aligning the different perspectives to avoid fragmentation of knowledge). This way, it gives engineers a formal way to break complexity down, communicate with stakeholders, and ensure consistency over the complete life cycle of a system. In this sense, layers and levels are not just ways to organize models, but they constitute the foundation that supports the design, analysis, and verification of modern complex systems.

### 2.5.1 System and Component Layers

The layered approach,shown in 2.12 is fundamental because modern system are too complex to be fully understood from a single viewpoint,the first layer is the operational layer ,which captures the system in its context of use. At this level, the system is not yet considered in terms of architecture or technology, but rather in term of the mission it must accomplish and the stakeholder needs it must satisfy. In this layer the main questions are: *What is the purpose of the system? Who are the stakeholders involved? Under which condition will the system be used?*.
Modeling at this layer elicits and structures stakeholder needs, mission objectives, and operational scenarios (normal, off-nominal, degraded). Typical artifacts include use-case models to delineate actor–system interactions; scenario and activity flows that narrate operational sequences; high-level state models for modes of operation; and a first set of operational requirements captured as model elements. Analysts also define preliminary Measures of Effectiveness (MoE) and constraints that will later guide design trade-offs. The aim is not to commit to any technical solution, but to produce a precise and testable statement of intent that will guide the rest of the engineering effort [19]. In SysML, this layer is often realized with Use Case Diagrams, Activity Diagrams for scenarios, and Requirement Diagrams to formalize needs and assumptions.
For example, in the automotive domain this could involve describing a vehicle's mission to transport passengers safely and efficiently, considering scenarios such as urban driving.By focusing on purpose rather than design, the operational layer creates a shared

understanding among stakeholders and establishes the foundation upon which all subsequent layers are built.

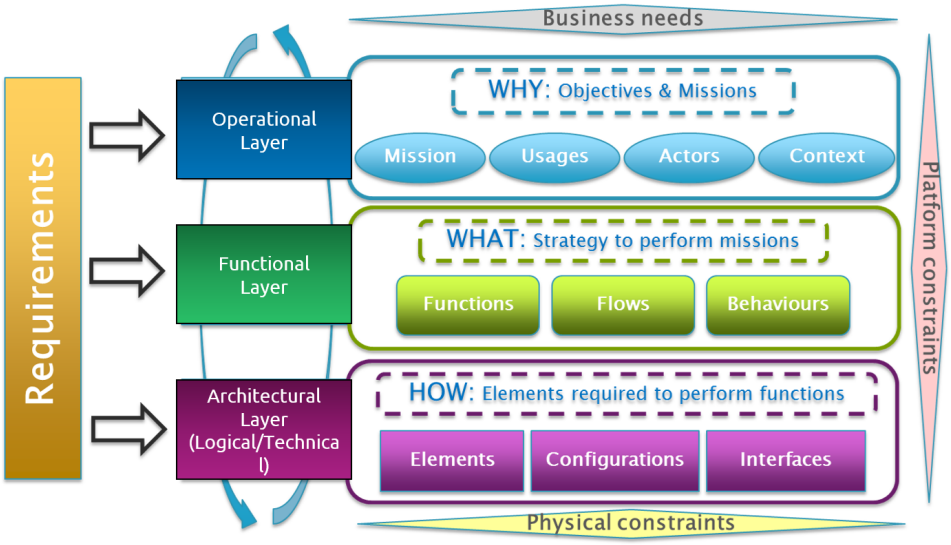


Figure 2.12: MBSE Layers

The second layer is functional or logical, which translates operational needs into functions and behaviors. Here, the system is described in terms of what it does rather what is made of. Functions are actions or transformations the system accomplishes to fulfill its intended objectives, while logical architectures describe the method of the functions and how they can interact with each other through flows and interfaces. This layer abstracts the physical implementation so engineers can objectively evaluate the various solution concepts without being confined by predetermined technologies. For example, *"ensuring passenger safety"* as an operational need could be expressed at the functional layer with functions such as *"detect obstructions," "assess collision potential," and "activate braking."* These functional requirements could later be developed in a number of ways (e.g., via radar sensors, camera sensors, or a combination of both), but the focus at this layer is to ensure the functional breakdown meets the mission needs. In this sense the functional layer transitions intentions to physical implementation, providing the logical architecture that will eventually be guided to allocate functions to physical components.
The modeling work being done here helps break down operational intent into functions and sub-functions, specify logical interfaces and flows (information, energy, material), and establish behavior using sequences, activities, and state machines. In the modeling process, engineers define the functional architecture, identify allocations from requirements to functions, and start to identify performance and parametric constraints as equations for early analysis. This abstraction intentionally protects design freedom, and allows for more systematic trade-off studies among competing concepts. In SysML, the modeling work represents behavior using typical Activity/Sequence Diagrams, defines the logical structure using Block Definition Diagrams (BDD), defines the interfaces using Internal Block Diagrams (IBD), and defines constraints and performance bounds using Parametric

Diagrams. The result is a technology-agnostic design specification that is fully traceable to the operational needs and ready for allocation to physical entities.
The third is physical, or implementation layer,specifies the with what the tangible realization of the logical design; defines the physical realization of the system. When you get to the physical layer, you are mapping logical functions and blocks to components, subsystems, and actual technologies. It describes how the system will be built, integrated, and maintained, also describes in detail all the software, hardware, and human-machine interfaces, allowing a clear physical basis for verification and validation to occur. By mapping each physical element to each function and permission, MBSE helps to ensure that no design action happens without regard for original intent.
Modeling focuses on allocating functions to hardware/software components, defining interfaces and protocols, and detailing integration architecture (e.g., mechanical assemblies, electronics, networks). Engineers refine non-functional requirements into component specifications, establish budgets (mass, power, timing, cost), and plan verification and validation by linking tests and test cases to the components and interfaces that implement each requirement. SysML BDD/IBD remain central for structure and interfaces; Requirement Diagrams capture derived and allocated specifications; Parametric Diagrams support performance compliance checks; and TestCase/«verify» relationships connect implementation to V&V evidence. Crucially, every physical element is tied back, via explicit model relations (e.g., «satisfy», «allocate», «verify»), to the logical functions and operational requirements it realizes, preserving end-to-end traceability.

| | Pillar | | | | |
|---|---|---|---|---|---|
| Layer of Abstraction | | Requirements | Behavior | Structure | Parametrics |
| | Operational Analysis | Stakeholder Needs | Use Cases | System context | Measurements of Effectiveness |
| | Functional Analysis | System Requirements | Functional Analysis | Logical Subsystems Communication | MoEs for Subsystems |
| | Technical Analysis | Component Requirements | Component Behavior | Component Assembly | Component Parameters |

Figure 2.13: MBSE Grid:Layer Analysis [5]

### 2.5.2 Abstraction and Model Levels

The layered perspectives discussed in the previous section highlight what aspects of a system can be represented in MBSE, ranging from operational intent to functional decomposition and physical realization. Yet, these perspectives alone are not sufficient to fully capture the complexity of modern systems. In addition to representing multiple viewpoints, systems must also be described across levels of abstraction, which reflect the hierarchical decomposition of a system into progressively finer elements. While layers separate different concerns, levels specify the granularity at which those concerns are modeled.

At the highest point of this hierarchy lies the system level, which considers the product as a whole in relation to its environment.At this level, the system is being treated as a complete whole that must satisfy the needs of the stakeholders, is subjected to external constraints, and must act as an integrated whole. For example, an automobile at the system level is modeled in terms of its overall mission safe and efficient transportation together with high-level requirements such as safety standards, fuel efficiency, or regulatory compliance. The system level provides the global context within which all lower levels must remain consistent.Just below system level, we have the subsystem level, which decomposes the system into large functional domains, or architectural 'blocks.' In the example of an automobile, this will include the powertrain, braking system, infotainment system, or driver assistance module. Each subsystem will be modeled with its own internal architecture and requirements, but always in relation to the system's high-level objectives. Subsystems are where cross-disciplinary interactions often become visible, requiring integration between mechanical, electrical, and software elements.

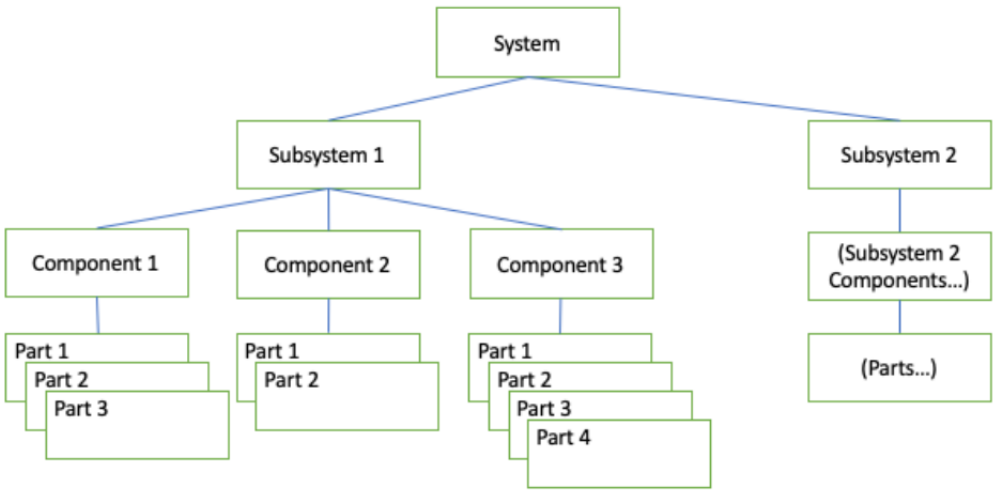


Figure 2.14: System Decomposition Diagram [6]

The next stage of decomposition is the component level, where subsystems are divided

into concrete functional units with well-defined interfaces. Continuing the automotive example, the braking subsystem might be realized through components such as the hydraulic actuator, brake pedal sensor, and electronic control unit (ECU). At this level, engineers specify design details, allocate functions to individual components, and begin to plan for integration and verification activities.Finally, at the most detailed stage lies the part level, which defines the individual physical elements such as microprocessors, circuit boards, sensors, and wiring harnesses that constitute the components. The part level is critical for manufacturing, testing, and logistics planning, and forms the bridge between system engineering models and product realization in practice.

### 2.5.3 System Requirement Definition

Having introduced the concepts of layers and levels, it is now possible to position the system requirements within this structured framework. Stakeholder needs are first formalized into requirements, which are then systematically allocated to the appropriate layer depending on their perspective,and are decomposed across the different levels of abstraction. Requirements act as the formal bridge between stakeholder needs and the technical specifications that ultimately shape the system architecture. According to ISO/IEC standards, a requirement can be defined as a *"statement that translates or expresses a need and its associated constraints and conditions"* [2]. Stakeholders are understood as individuals or organizations with a legitimate interest in the system, ranging from customers and regulatory authorities to manufacturers, suppliers, and end-users. Each stakeholder introduces a unique set of expectations, which must be translated into requirements that are clear, verifiable, and technically actionable [12].
At a high level, there is a distinction between stakeholder requirements and system requirements. Stakeholder requirements are informal expressions of desired capabilities, usually in natural language, and consequently open to interpretation. System requirements distill these informal expressions into structured and formal technical statements that can shape their design and implementation. To accommodate varying degrees of specificity in system requirements, there are often splits within system requirements between functional requirements that describe what the system must do, non-functional requirements that capture quality attributes such as safety, reliability, and performance, and constraints that detail technological, regulatory, or operational restraints. [2] This structured hierarchy ensures all design decisions can be traced back to original stakeholder needs which allows for consistency and accountability in a system during its life cycle.
An essential principle of this process is traceability, which refers to the clear linkage of requirements to functions, subsystems, components and ultimately to manufactured parts. This effectively ties any failure of a physical element to the requirement that defined its inclusion. For an aircraft, for instance, the requirement *"the system shall be capable of propulsion,"* can ultimately be linked back to the intended physical component through the envisioned functionality (linking it to the function "propel") and logical block (e.g. "engine") that should eventually manifest to a physical component (e.g. turbofan). Thus, traceability purposes to decompose system complexity into manageable layers and gives assurance that all needs have corresponding solutions, while all solutions can be linked

back to a higher-level corresponding need. [2]
The quality of requirement statements is critical to the success of the entire development process. As emphasized by INCOSE, requirement statements must adhere to specific rules to ensure that they are clear, complete, consistent, verifiable, and achievable [13]. INCOSE also recommends the use of attributes such as requirement ID, priority, verification method, and responsible stakeholder to support requirement management. Poorly formulated requirements often lead to costly rework and system failures, whereas structured, high-quality statements create the conditions for effective design and validation. Patterns for requirement statements are also encouraged, with structures varying depending on the requirement type (e.g., functional, performance, interface),an illustrative example is reported in Table 2.1. These patterns serve to reduce ambiguity and enforce discipline in the definition of requirements [12].

| **Type** | **Definition and Pattern** |
|---|---|
| Functional Requirement | Define what functions need to be performed to accomplish the objectives.<br>*The SYSTEM shall exhibit FUNCTION while in CONDITION.* |
| Performance Requirement | Define how well the system needs to perform the functions.<br>*The SYSTEM shall FUNCTION with PERFORMANCE and TIMING upon EVENT TRIGGER while in CONDITION.* |
| Design Constraint Requirement | Limit the options open to a designer of a solution by imposing immovable boundaries and limits.<br>*The SYSTEM shall exhibit DESIGN CONSTRAINTS in accordance with PERFORMANCE while in CONDITION.* |
| Environmental Requirement | Define which characteristics the system should exhibit when exposed in specific environments.<br>*The SYSTEM shall exhibit CHARACTERISTIC during or after exposure to ENVIRONMENT for EXPOSURE DURATION.* |
| Suitability Requirement | Include a number of the "-ilities" (qualities) in requirements to include.<br>*The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION for CONDITION DURATION.* |

Table 2.1: Types of requirements and their definition/patterns [12]

In the context of MBSE, requirements are not merely documented but formally modeled. Tools such as Cameo Systems Modeler support the creation of requirement diagrams, allocation tables, and traceability matrices that link requirements to functions and design elements. Frameworks such as MagicGrid provide methodological guidance, ensuring that requirements are progressively refined and connected across multiple modeling layers. This approach extends the traditional process by embedding requirements within

a digital model, where they can be validated, verified, and even connected directly to optimization workflows such as those implemented through MDO. The integration of requirements within MBSE environments thus strengthens the digital continuity of the system development process, ensuring that stakeholder needs are preserved and traceable from conception to validation and beyond.

## 2.6 SysML

Considering the challenges posed by the growing complexity of modern systems, it becomes essential to complement MBSE with a formal language that ensures consistency and traceability across requirements, architectures, and behaviors. Traditional document-based approaches lack this formal rigor, often leading to fragmented system descriptions and loss of consistency across the life cycle. To overcome these limitations, the Systems Modeling Language (SysML) has emerged as the de facto standard for representing system models within the MBSE paradigm.
SysML has been developed on the basis of the Unified Modeling Language (UML) a general-purpose modeling language that was first crafted in the 1990s to be a standard approach for software design.(2.15) UML standards and practices are now widely utilized in software engineering to provide a graphical notation for specifying, visualizing, and documenting software-intensive systems. UML focuses on modeling classes and behavior, as well as their interactions, and is certainly powerful for software-centric applications. However, when it comes to more focused or foundational views, UML leaves a lot outside its domain and does not consider the hardware, human operators and processes, and other organizational constraints that must complement any software in the scope of systems engineering. [25, 26] SysML was standardized as a profile and extension of UML

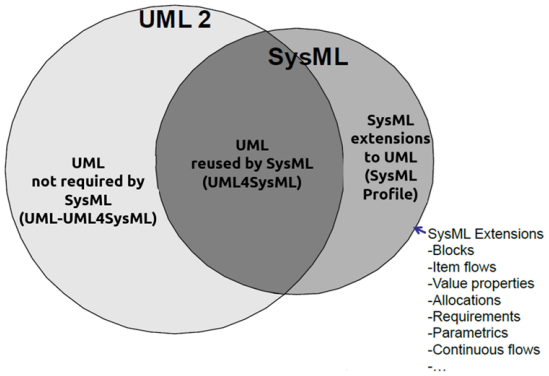


Figure 2.15: SysML and UML relationship [7]

through the Object Management Group (OMG) to support systems engineering needs. SysML simplifies certain parts of UML and extends other parts to provide a modeling language that integrates structural, behavioral, requirement, and parametric representations of a system. In this respect, SysML is a domain-independent language because it can model not only software, but also physical architectures, logical architectures, and

others. SysML thereby gives multidisciplinary teams a common modeling language that can communicate across domains and maintain the consistency among operational, functional, and physical levels and among the different systems levels. [27] The distinction

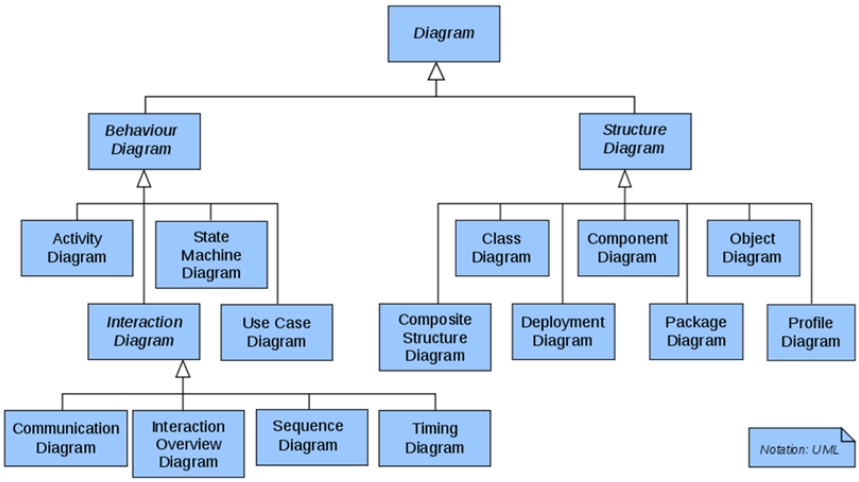


Figure 2.16: UML Diagrams

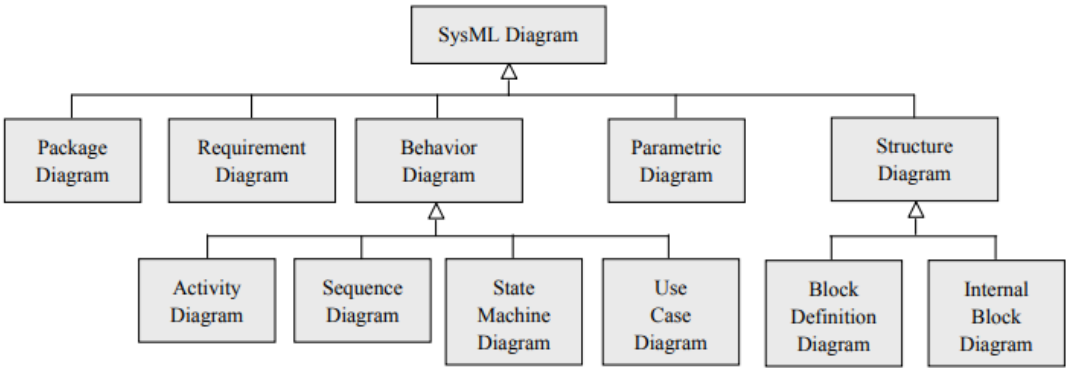


Figure 2.17: SysML Diagrams [8]

between UML and SysML can be summarized across several dimensions. UML comprises 13 diagram types(2.16), heavily oriented toward software structure and behavior, whereas SysML employs 9 diagram types,(2.17), that span four complementary categories: requirements, structure, behavior, and parametrics. Unlike UML, SysML treats requirements as first-class model elements, linking them explicitly to design and verification artifacts. Furthermore, SysML's parametric diagrams provide a formal means to

represent mathematical constraints and performance models, enabling engineers to integrate analytical reasoning directly into system models. In this way, SysML extends the modeling paradigm beyond software to capture the multidisciplinary nature of complex engineered systems. This categorization reflects the multidimensional nature of MBSE: requirements define what must be achieved, structures describe what the system is made of, behaviors explain how it operates over time, and parametrics capture how performance is quantified and constrained.

- **Requirements diagrams**: These diagrams allow requirements to be modeled as explicit elements within the system model, instead of being managed only in textual documents. Requirements can be hierarchically organized, refined, and linked to design elements using relationships such as «satisfy», «verify», and «refine». This enables engineers to establish end-to-end traceability from stakeholder needs to system architecture and validation tests.

- **Structural diagrams**: SysML employs structural diagrams to define the static architecture of a system. The Block Definition Diagram (BDD) captures the decomposition of the system into blocks and subsystems, showing composition, specialization, and dependency relationships. Blocks can represent not only software modules but also physical hardware or even abstract concepts such as interfaces. Complementing the BDD, the Internal Block Diagram (IBD) specifies the internal structure of a block by showing its parts, ports, and the flows exchanged through connections. In the automotive example, a BDD might show the decomposition of the vehicle into blocks such as Powertrain, Chassis, and Infotainment System, while an IBD of the Powertrain block could detail the interactions between the engine, transmission, and control units. Together, BDD and IBD provide a robust structural framework for integrating physical and logical architectures.

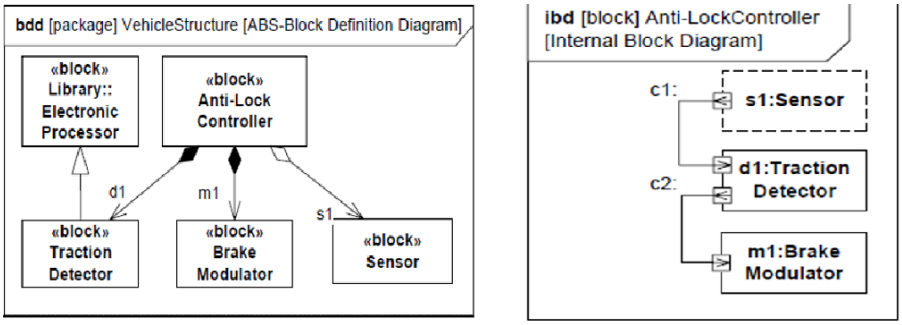


Figure 2.18: IBD and BDD Diagrams

- **Parametric diagrams** are unique to SysML and provide a means to model quantitative relationships and constraints. They link parameters of blocks using constraint equations, enabling integration with simulation and analytical tools. This makes it possible to verify whether the system meets performance requirements early in the design process. For instance, in a vehicle's thermal management system, a

parametric diagram could relate variables such as coolant flow rate, heat generation allowing engineers to analyze whether thermal requirements are satisfied under different operating conditions.

- **Behavioral diagrams** capture the dynamic aspects of a system, describing how it responds to stimuli, executes functions, and interacts over time.These diagrams offer complementary perspectives on system behavior, each focusing on a different dimension of dynamics. At a high level, use case diagrams provide an intuitive entry point by identifying the actors that interact with the system and the functional services offered to them. They are particularly useful during early stages of analysis, when stakeholder needs and high-level capabilities are being defined. Once these capabilities have been established, more detailed behavioral models refine the description of how functions unfold over time. Activity diagrams are commonly used to represent workflows and control flows, describing sequences of actions and the data or control dependencies among them. They are well suited for modeling operational processes. Sequence diagrams, in contrast, emphasize the temporal dimension of behavior by illustrating the exchange of messages or signals between system elements. They are particularly effective for describing scenarios of interaction. State machine diagrams complement these perspectives by focusing on the different states that a system or component may assume and the transitions triggered by events.

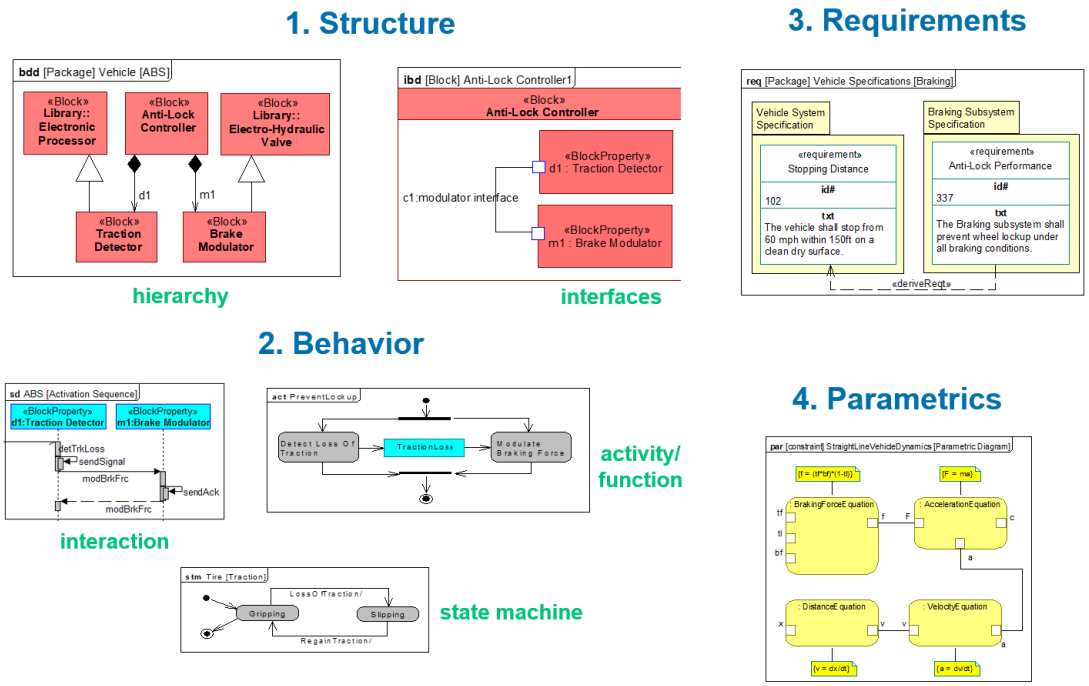


Figure 2.19: SysML Diagrams

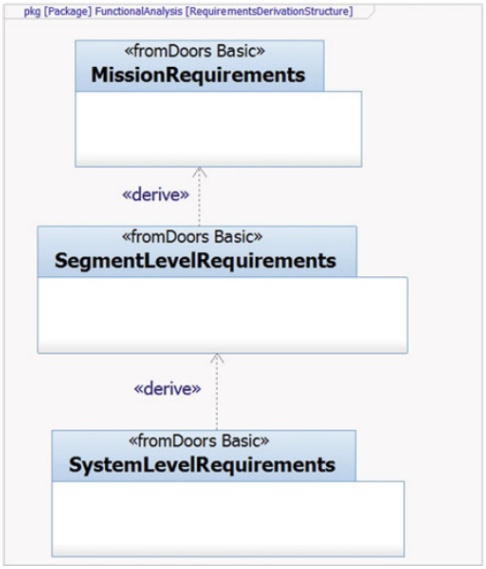


Figure 2.20: Example of package Diagram [2]

The integration of requirements, structural, behavioral, and parametric perspectives within a single language makes SysML a powerful enabler of MBSE practices. By embedding requirements directly into the model and linking them to design and verification artifacts, SysML ensures end-to-end traceability, thereby reducing the risk of overlooked dependencies and facilitating systematic validation. Its ability to represent both logical abstractions and physical implementations provides a natural mechanism for consistency across layers and levels, ensuring that design intent remains aligned with stakeholder needs. Moreover, the explicit modeling of quantitative relationships through parametric diagrams supports early trade-off analyses and performance verification, reducing costly rework in later stages of development. SysML also fosters multidisciplinary communication, since engineers, software developers, and domain experts can work within a common semantic framework rather than relying on heterogeneous document sets.
Taken together, these features make SysML more than a graphical notation: it is a formalized language for systems engineering, designed to capture the complexity of modern products and support their evolution throughout the life cycle. By providing an integrated modeling environment, SysML transforms MBSE from a conceptual methodology into a practical, executable approach, enabling organizations to manage complexity, accelerate development, and increase confidence in system outcomes. In this sense, SysML serves both as a technical tool and as a strategic enabler, bridging the gap between stakeholder needs, engineering design, and system validation. [19]

## 2.7 MDO

### 2.7.1 Introduction

The design of contemporary engineering systems is creating a need for combining knowledge from many fields. Traditional methods have generally involved optimizing single disciplines in isolation or sequentially, as opposed to holistically and concurrently, will not suffice for the complexity that tightly coupled architectures continually present. What may be deemed an optimal solution from one discipline's perspective may not be optimal, or even feasible, when the interactions of other disciplines are included. Multidisciplinary Design Optimization (MDO) can assist with these challenges. MDO represents a methodological framework for modeling, analyzing, and optimizing multiple disciplines in a single integration process.

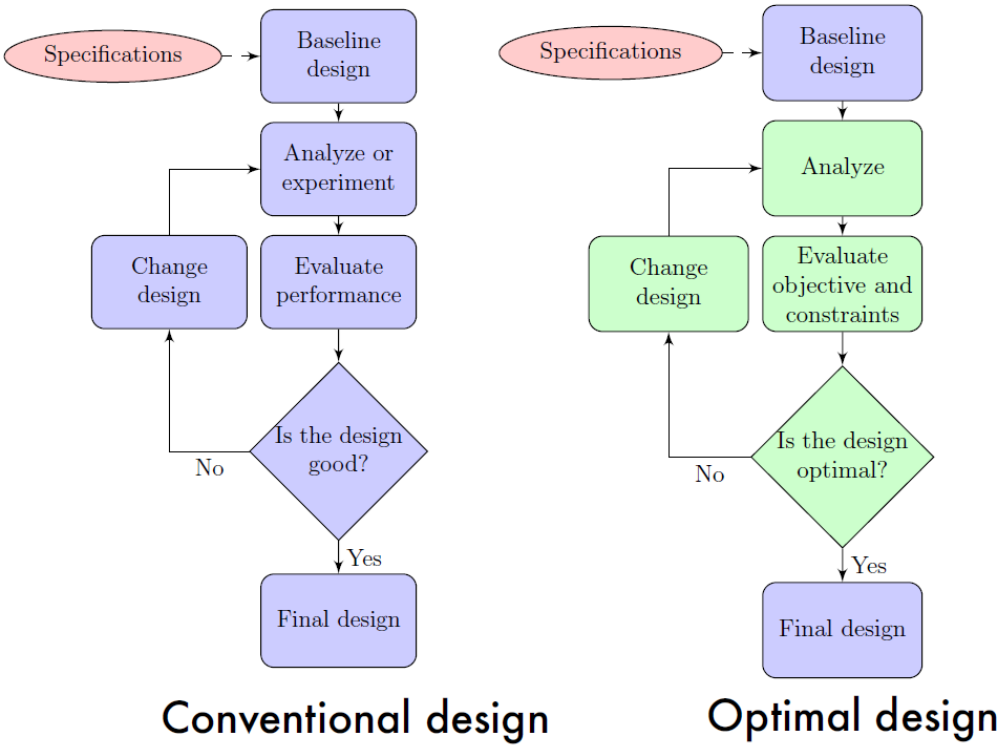


Figure 2.21: Comparison between conventional design and optimal design [9]

MDO can be defined as the systematic application of optimization techniques to the design of systems involving strongly coupled disciplinary analyses, with the aim of achieving globally optimal solutions that satisfy requirements across all domains [10]. Unlike conventional design approaches, where disciplines are often considered independently, MDO explicitly accounts for cross-disciplinary dependencies, thereby enabling the structured

exploration of trade-offs and synergies. Its origins are closely tied to aerospace engineering, where aerodynamic, structural, and control disciplines are intrinsically interdependent in aircraft design [9]. Over time, its scope has expanded to domains such as automotive, energy, and space systems, reflecting the growing need for integrated optimization in the design of complex architectures [12, 28].
At its core, an MDO problem can be cast as a constrained optimization problem that involves a set of design variables, objective functions, and constraints. Design variables may be distributed across disciplines and constraints typically reflect physical laws, performance limits, or stakeholder needs. Objective functions may represent performance measures, such as weight, cost, or efficiency that must be optimized simultaneously and in the context of disciplinary coupling. The real difference in multi-disciplinary optimization comes, not from the optimization, but from the coordination made possible when analyses for the disciplinary aspects are coupled in the optimization loop.(2.21) An optimizer at the system level must perform a coordinated set of evaluations from analysis codes for a wide range of disciplines, determines that the global solution respects all the conditions where disciplinary information must be consistent, and converge to an optimal design that satisfies a set of local and system-wide requirements. [10]
A central concept in MDO is that of architectures, which define how the optimization process is organized relative to the disciplinary analyses. Architectures establish the flow of information among system and disciplinary models, allocate responsibilities between optimizers, and determine how multidisciplinary consistency is enforced. As Martins and Lambe emphasize, the choice of architecture is often as critical as the mathematical formulation of the optimization problem itself, since it strongly influences computational cost, scalability, and ease of implementation. [10]
Among the various approaches developed in the literature, the simplest class is represented by the monolithic architectures.Monolithic architectures are of particular importance for two reasons. First, they provide the theoretical foundation for understanding MDO, since most alternative formulations can be derived from a monolithic framework by introducing decomposition or relaxation strategies. Second, they serve as benchmarks: their conceptual simplicity and rigorous mathematical basis make them useful references for evaluating the efficiency and scalability of more advanced approaches. [10]
The principal monolithic formulations discussed in the literature are All-at-Once (AAO), Multidisciplinary Feasible (MDF), Individual Discipline Feasible (IDF) and Simultaneous Analysis and Design (SAND) architectures. Each of these represents a different balance between problem size, computational cost, and the manner in which disciplinary consistency is enforced.

### 2.7.2 The All-at-Once (AAO) Problem

The most general way to formulate an MDO problem is the *All-at-Once (AAO)* architecture. In this approach, all design variables, state variables, and coupling variables including their target copies are treated as optimization variables. The disciplinary residual equations, physical constraints, and consistency constraints are all enforced explicitly within the optimization problem. The AAO formulation can be expressed as:

$$\begin{aligned}
\text{minimize} \quad & f_0(x,y) + \sum_{i=1}^{N} f_i(x_0, x_i, y_i) && (2.1)\\
\text{with respect to} \quad & x, y^t, y, \overline{y} && (2.2)\\
\text{subject to} \quad & c_0(x,y) \geq 0, && (2.3)\\
& c_i(x_0, x_i, y_i) \geq 0, \quad i = 1, \ldots, N, && (2.4)\\
& c_i^c = y_i^t - y_i = 0, \quad i = 1, \ldots, N, && (2.5)\\
& R_i(x_0, x_i, y_{j\neq i}^t, y_i, y_i) = 0, \quad i = 1, \ldots, N. && (2.6)
\end{aligned}$$

where:

| Symbol | Definition |
|---|---|
| $x$ | Vector of design variables |
| $y^t$ | Vector of coupling variable targets (inputs to a discipline analysis) |
| $y$ | Vector of coupling variable responses (outputs from a discipline analysis) |
| $\bar{y}$ | Vector of state variables (variables used inside only one discipline analysis) |
| $f$ | Objective function |
| $c$ | Vector of design constraints |
| $c^c$ | Vector of consistency constraints |
| $\mathcal{R}$ | Governing equations of a discipline analysis in residual form |
| $N$ | Number of disciplines |
| $n_{()}$ | Length of given variable vector |
| $m_{()}$ | Length of given constraint vector |
| $()_0$ | Functions or variables that are shared by more than one discipline |
| $()_i$ | Functions or variables that apply only to discipline $i$ |
| $()^*$ | Functions or variables at their optimal value |
| $\tilde{()}$ | Approximation of a given function or vector of functions |

Figure 2.22: Mathematical notation for MDO problem formulations [10]

In the field of MDO, XDSM stands for Extended Design Structure Matrix. It is a graphical representation method used to describe the process flow of information between analysis disciplines, optimization algorithms, and system-level models. By explicitly illustrating data dependencies and the sequence of computational tasks, it provides a clear and standardized way to document, analyze, and communicate MDO architectures. [10] To keep the diagrams compact,it is assumed by convention that any block referring to discipline

i represents a repeated pattern for every discipline. Thus, in Figure 2.23 a residual block exists for every discipline in the problem and each block can be executed in parallel. As an added visual cue in the XDSM, the *"Residual i"* component is displayed as a stack of similar component.
Although AAO provides the most general and flexible problem formulation, it is rarely solved directly in practice due to the large number of optimization variables and constraints. Instead, AAO is often used as a conceptual starting point, from which other monolithic formulations can be derived by removing groups of constraints.

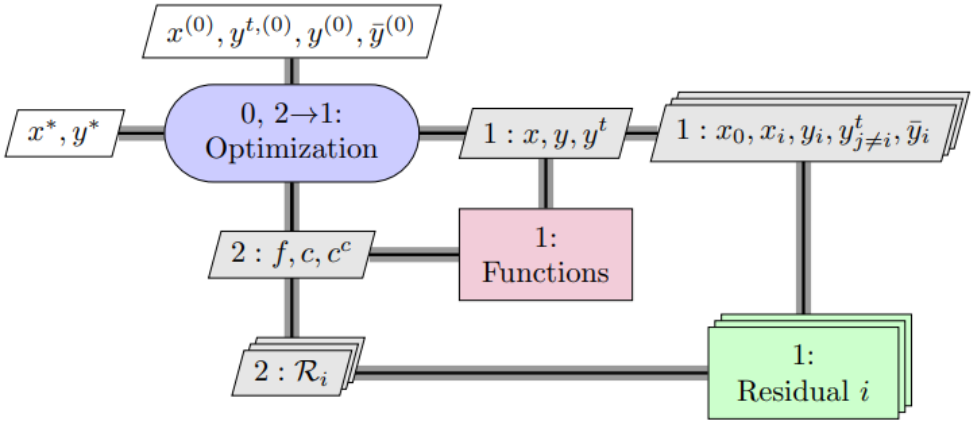


Figure 2.23: XDSM for solving the AAO problem [10]

### 2.7.3 The Multidisciplinary Feasible (MDF) Problem

The *Multidisciplinary Feasible (MDF)* architecture is obtained from AAO If both analysis and consistency constraints are removed from Problem. Feasibility across disciplines is instead enforced by solving a complete *Multidisciplinary Analysis (MDA)* at each iteration. This architecture has also been referred to in the literature as Fully Integrated Optimization and Nested Analysis and Design. [10] The resulting optimization problem is:

$$\begin{aligned}
\text{minimize} \quad & f_0(x, y(x, y)) && (2.7)\\
\text{with respect to} \quad & x && (2.8)\\
\text{subjected to} \quad & c_0(x, y(x, y)) \geq 0, && (2.9)\\
& c_i(x_0, x_i, y_i(x_0, x_i, y_{j\neq i}) = 0, \quad i = 1, \ldots, N, && (2.10)
\end{aligned}$$

where $y(x)$ are the coupling variables obtained from solving the MDA for a given set of design variables. MDF is conceptually simple and requires the smallest optimization

problem, expressed only in terms of design variables. However, it requires solving the full MDA at every iteration, which is computationally expensive [9, 10].

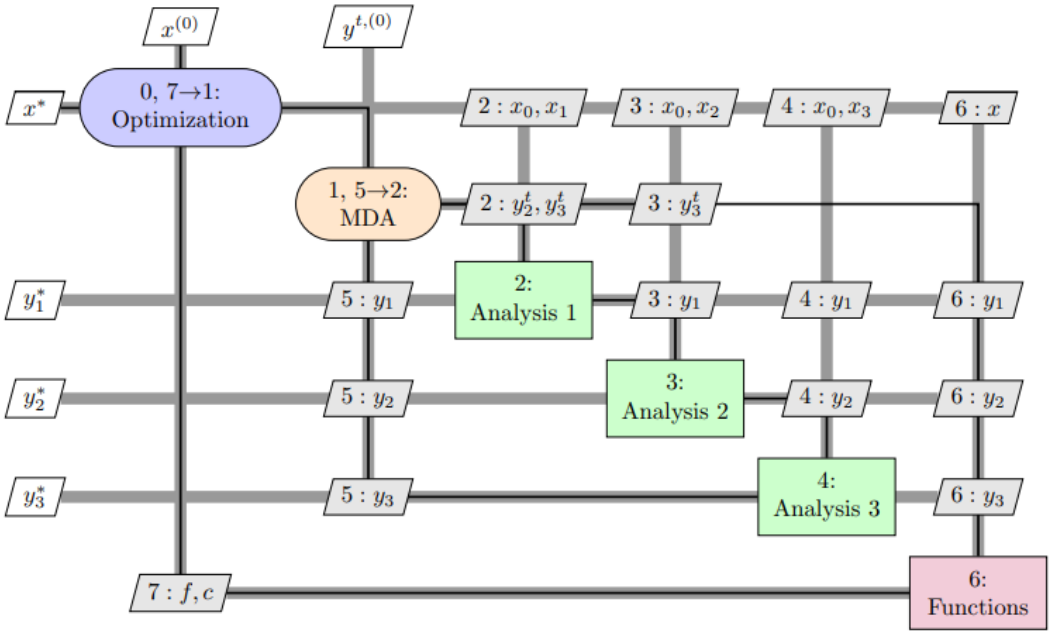


Figure 2.24: XDSM for solving the MDF problem [10]

### 2.7.4 The Individual Discipline Feasible (IDF) Problem

The *Individual Discipline Feasible (IDF)* is obtained by eliminating the disciplinary analysis constraints $Ri(x_0, x_i, y_i, y^t_{j\neq i}, \overline{y_i}) = 0$, it avoids solving a full MDA at each iteration by introducing *target variables* as optimization variables. Each discipline is solved independently, and consistency is enforced by adding equality constraints to match target and actual coupling variables. The formulation is:

$$\begin{aligned}
\text{minimize} \quad & f_0(x, y(x, y^t)) && (2.11)\\
\text{with respect to} \quad & x, y^t && (2.12)\\
\text{subjected to} \quad & c_0(x, y(x, y^t)) \geq 0, && (2.13)\\
& c_i(x_0, x_i, y_i(x_0, x_i, y^t_{j\neq i}) = 0, \quad i = 1, \ldots, N, && (2.14)\\
& c^c_i = y^t_i - y_i(x_0, x_i, y_i(x_0, x_i, y^t_{j\neq i}) = 0, \quad i = 1, \ldots, N, && (2.15)
\end{aligned}$$

A key advantage of IDF is that it enables the use of existing disciplinary codes with minimal modification, since they can be treated as independent black-box solvers. This also makes it possible to execute analyzes in parallel, with consistency across disciplines

enforced through target variables and explicit consistency constraints.Nevertheless, challenges remain. If the number of coupling variables is large, the optimization problem can still grow significantly, limiting efficiency. Moreover, for gradient-based methods which are often essential in large scale optimization the computation of consistent gradients becomes a major difficulty. Gradients must reflect discipline feasibility, and inaccuracies in their evaluation can severely degrade optimizer performance. Thus, while IDF strikes a balance between computational tractability and compatibility with existing analysis tools, it still faces scalability and sensitivity challenges in practice [10].

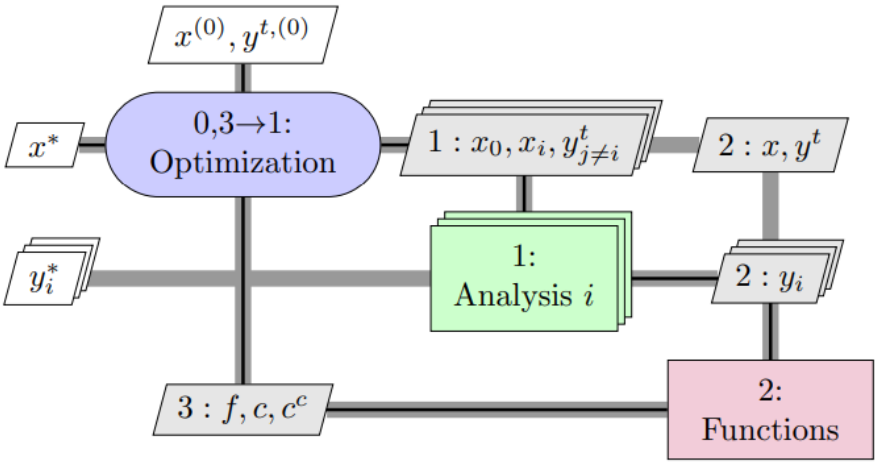


Figure 2.25: XDSM for solving the IDF problem [10]

### 2.7.5 The Simultaneous Analysis and Design (SAND) Problem

The *Simultaneous Analysis and Design (SAND)* architecture is obtained by simplifying the AAO formulation through the elimination of consistency constraints, merging target and response variables into a single set of coupling variables. In this way, the optimizer directly handles both design and analysis variables, simultaneously exploring design and feasibility.The formulation is:

$$\begin{aligned}
\text{minimize} \quad & f_0(x, y) && (2.16)\\
\text{with respect to} \quad & x, y, \overline{y} && (2.17)\\
\text{subjected to} \quad & c_0(x, y) \geq 0, && (2.18)\\
& c_i(x_0, x_i, y_i) = 0, \quad i = 1, \ldots, N, && (2.19)\\
& R_i(x_0, x_i, y, \overline{y}_i) = 0, \quad i = 1, \ldots, N, && (2.20)
\end{aligned}$$

A key feature of SAND is that it does not require explicit or exact solution of the disciplinary analyses at every iteration. This allows the optimizer to traverse infeasible

regions of the design space, potentially accelerating convergence. Moreover, SAND is not restricted to multidisciplinary contexts and can be applied to single-discipline optimization, where it reduces to the familiar case of PDE-constrained optimization.
However, two major drawbacks limit its practical use. First, the problem formulation still requires the inclusion of all state variables and residual equations, leading to very large problem sizes and the risk of the optimizer stalling at infeasible designs. Second, since disciplinary residuals must be expressed explicitly as constraints, they and their derivatives must be made available to the optimizer. In practice, many engineering codes operate as "black boxes" and do not expose residuals or internal state variables, making the implementation of SAND often impractical. For this reason, while SAND remains conceptually important, most engineering applications prefer alternative formulations such as MDF or IDF, which are more compatible with existing disciplinary solvers. [10]

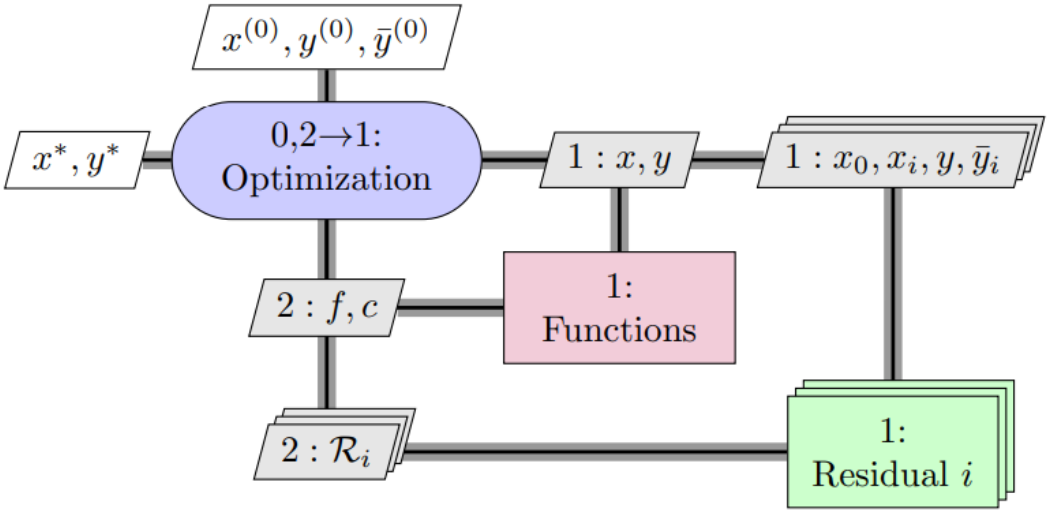


Figure 2.26: XDSM for solving the SAND problem [10]

### 2.7.6 Comparison of Monolithic Architectures

When comparing the monolithic architectures of multidisciplinary design optimization (MDO), it is clear that each formulation represents a different compromise between problem size, feasibility enforcement, and compatibility with existing disciplinary analyses. The AAO formulation serves mainly as a theoretical starting point, while MDF, IDF, and SAND offer more practical alternatives. MDF guarantees multidisciplinary consistency but requires repeated costly analyses; IDF introduces target variables to allow parallel execution while increasing problem size; and SAND avoids nested analyses but relies on explicit access to residual equations, which limits applicability in real engineering contexts. A comparative summary is provided in Table 2.2.

| **Architecture** | **Advantages** | **Disadvantages** |
|---|---|---|
| **AAO** | Most general and flexible formulation; includes all design, state, and coupling variables; serves as theoretical starting point for deriving other formulations. | Very large optimization problem; rarely solved in practice; high computational burden. |
| **MDF** | Simplest and most intuitive approach; optimization problem expressed only in design variables; smallest problem size; intermediate designs always feasible. | Requires a full multidisciplinary analysis (MDA) at every iteration; derivative computation is costly; poor scalability for large systems. |
| **IDF** | Avoids nested MDA by introducing target variables; allows disciplines to run independently and in parallel; good balance between feasibility and flexibility. | Problem dimension increases due to target variables and consistency constraints; slower convergence possible. |
| **SAND** | Eliminates nested analyses; feasibility enforced directly by optimizer; can accelerate convergence for small/medium-scale problems. | Optimization problem grows significantly by including all state variables; limited scalability for large systems. |

Table 2.2: Comparison of monolithic MDO architectures [9, 10, 12].

## 2.8 Integration MBSE and MDO

Modern engineering systems have become more complex as technology and their relationship with humanity evolve. Engineering systems today require methods that are capable of structuring information as well as supporting decisions based on quantitative analysis. In this context two complementary approaches were developed: MBSE and MDO. While they come from different roots, their potential together provide a very robust method of design and validation for complex systems.
MBSE emphasizes representing and managing the system through its life cycle using formal models rather than the conventional document-based paradigm. MBSE is a development method that employs modeling languages such as SysML to use models to capture requirements, system architectures, interfaces, and behavioral scenarios entirely in a unified framework. The focus of MBSE is not on numerical optimization but to create traceability, consistency, and communication with the stakeholders and engineering teams' designs and engineering activities. MBSE, in this way, serves as the semantic foundation that connects and aligns stakeholder requirements, design artifacts, and testing/verification activities, which helps prevent inconsistencies or integration errors.
On the other hand, MDO mainly focuses on the optimization or trade space of engineering systems with multiple interdependent disciplines. MDO focuses on an area for more optimized designs through multi-domain analysis and optimization of the design, look at the tradeoffs between each discipline, and discounted individual discipline optima in favor of a global optimum. MDOs benefit is navigating the system towards less cost, less mass, less development and more performance than found through only simply optimizing an individual discipline. Therefore the use of MDO is a computational engine for identifying alternatives and helping the system work towards optimal with quantitative results. MBSE and MDO are separate but complementary tools: MBSE provides a formal representation and traceable design space, MDO branches out into the design space to discover solutions it could identify. The combined use is the natural next step of further complication and resolution of performance.


Figure 2.27: Tools used for the integration

The differences between MBSE and MDO are also apparent in the tooling that is available for use. For example, in the MBSE domain, one of the most well-known environments is Cameo Systems Modeler. Cameo is a full environment for MBSE and is developed with the SysML language. Using Cameo means that engineers can capture system requirements, develop the functional architecture and physical architecture, develop parametric

relationships, and create traceability across the various layers of the system model. In addition to modeling, Cameo supports simulation and validation so that engineers are able to demonstrated early verification that the system design will satisfy the requirements. One of the best capabilities of Cameo is that engineers can manage complexity through structured diagrams and the life cycle is well documented through traceability so that if the requirements or design changes, the model can propagate that change across the model.
In contrast, MDO environments are focused on numerical computation and optimization. For example, GEMSEO is an open-source Python package specifically designed for multidisciplinary optimization. GEMSEO defines MDO problems, connects disciplinary analyzes, and utilizes specific optimization architectures like MDF, IDF, or SAND. Since it is written in Python, it works smoothly with numerical solvers and external simulation tools, allowing engineers to automate workflows and run design studies more efficiently. One of its main advantages is that it can use existing disciplinary models and connect them into a single integrated optimization process.
These two toolchains highlight the different philosophies of MBSE and MDO: MBSE tools such as Cameo emphasize formal representation, communication, and traceability, while MDO tools such as GEMSEO prioritize numerical analysis, optimization, and computational efficiency. Bridging these two worlds therefore requires both conceptual alignment and technical integration, ensuring that the structured models developed in MBSE can be translated into the computational workflows required for MDO.

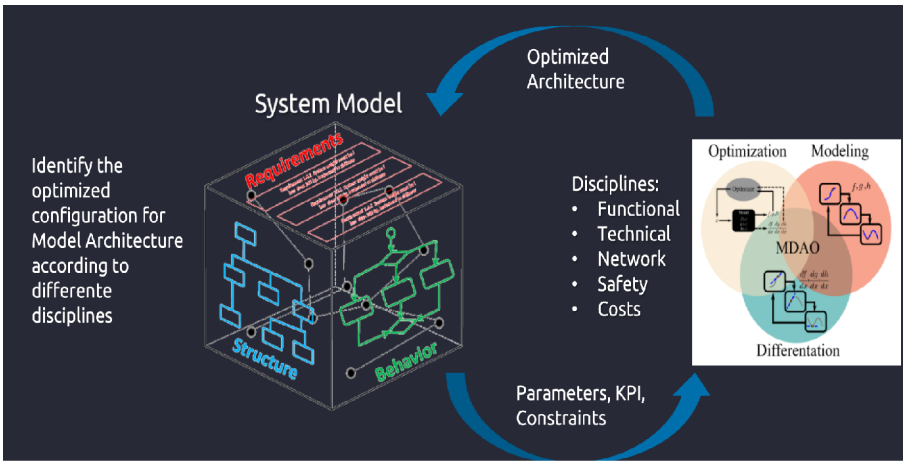


Figure 2.28: MBSE and MDO integration

The overarching objective of this work is to define the electronic architecture of an automobile and to determine its most suitable configurations with respect to two critical criteria: cost and weight. The challenge lies in exploring multiple possible ECU allocations and identifying the solutions that optimally balance performance and resource constraints. Addressing this problem requires not only a systematic representation of the system architecture but also a quantitative optimization process capable of evaluating

trade-offs across many alternatives.
The integration between MBSE and MDO has been realized through the combined use of Cameo Systems Modeler and GEMSEO in Python. As said before Cameo is used to model the system architecture, requirements, and parametric relationships of the automotive electronic system. Within Cameo, dedicated tables are created to capture relevant design data such as ECU functions. These tables can be exported in a CSV format, which serves as the interface between the MBSE environment and the optimization framework. On the MDO side, Python scripts leveraging GEMSEO can directly import these CSV files and transform them into structured input data for the optimization process. In this way, the inputs for the optimization problem are generated directly in Cameo, ensuring full traceability from system requirements to design parameters. The optimization then proceeds by applying the selected MDO architecture and algorithms to explore alternative configurations and identify optimal solutions.
The outputs of the process, as presented in this thesis, are therefore already optimized configurations of the electronic control units, evaluated in terms of cost and weight. This integration establishes a seamless workflow: Cameo provides the formal and traceable definition of the system, while GEMSEO acts as the computational engine that delivers quantitatively superior solutions. The result is a digital process that bridges descriptive system modeling and optimization-driven design exploration.

# Chapter 3

# Case study

## 3.1 The e-MBE Initiative at Capgemini: Structure, Objectives, and Team Contributions

Capgemini is a global consulting company that operates at the intersection of technology, engineering, and digital transformation. With a consolidated presence in the automotive, aerospace, and industrial sectors, the company supports organizations in adopting innovative methodologies and tools to improve efficiency, quality, and competitiveness. Within this framework, Capgemini has established several Centers of Excellence, designed to develop and disseminate best practices in advanced engineering disciplines.
During my internship, I was part of the Center of Excellence for Systems Engineering and Product Design, which plays a central role in promoting the adoption of model-based approaches within complex industrial projects. In particular, I contributed to the e-MBE (Extended Model-Based Engineering) project, an internal initiative aimed at extending and industrializing MBSE practices across different phases of the product life cycle. The project is structured around three specialized teams shown in 3.1, each with distinct goals and complementary expertise.

- The first stream, called **Model-Based Digital Continuity**, focuses on establishing a model-based approach to ensure data and process continuity across the product life cycle. Its objectives include the development of a digital thread architecture and the integration of MBSE with CAD and software environments. By doing so, the team ensures that product information remains consistent and traceable across design domains and throughout successive development phases.

- The second stream, known as **Early Verification and Validation (V&V)**, is dedicated to improving the validation of requirements and functions during the early phases of development. Its activities involve the preliminary validation of requirements, functional simulation of early designs, prototyping of human–machine interfaces (HMI) during the concept stage, and the definition and validation of test cases. The goal is to detect inconsistencies and potential issues at an early stage, thereby reducing costly design changes later in the process.

- The third stream, **Advanced Decision Making**, is the one I actively participated in. This team addresses the need for robust methodologies to support strategic design decisions in the early phases of system development. Its objectives are threefold: first, to optimize system architectures at the earliest design stages, when the potential for improvement is highest; second, to provide robust decision-making methods that can account for uncertainty and variability; and third, to explore the integration of MBSE and MDO as a means of enhancing decision quality. By connecting the descriptive power of MBSE with the quantitative rigor of MDO, this group aims to provide decision-makers with reliable, traceable, and optimized design alternatives that can significantly improve the overall system development process.

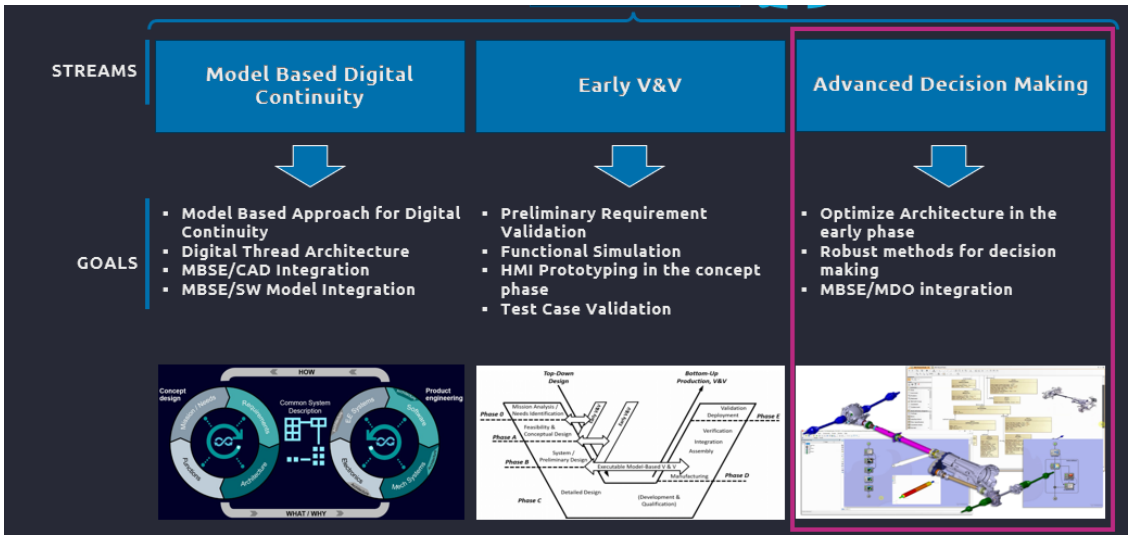


Figure 3.1: MBSE and MDO integration

## 3.2 Case study overview

The case study presented in this thesis arises from a concrete industrial challenge that affects the automotive sector: the need to reduce both the cost and the complexity of the electronic architectures of modern vehicles. Over the past decades, the number of electronic control units integrated into automobiles has steadily increased to support a wide range of functions, from basic comfort features to advanced safety systems and connectivity services. While this expansion has enhanced vehicle performance and user experience, it has also generated several drawbacks. The proliferation of ECUs results in higher production and integration costs, redundant wiring, greater system weight, and a rise in overall complexity that makes verification and maintenance more challenging. As a consequence, car manufacturers are actively exploring systematic approaches to rationalize ECU allocation and to design electronic architectures that remain cost-efficient, lightweight, and easier to manage.

The objective of this thesis is to demonstrate how the integration of Model-Based Systems Engineering and Multidisciplinary Design Optimization can provide a methodological and practical framework to address this challenge. More specifically, the goal is to define an electronic architecture for an automobile,like in figure 3.2, and to identify its most suitable configurations with respect to two primary criteria: cost and weight. To delimit the scope of the study, the project focuses on the allocation of functions belonging to three subsystems that are representative of typical body electronics: the door locking and unlocking system, the climate control system, and the window opening and closing system.

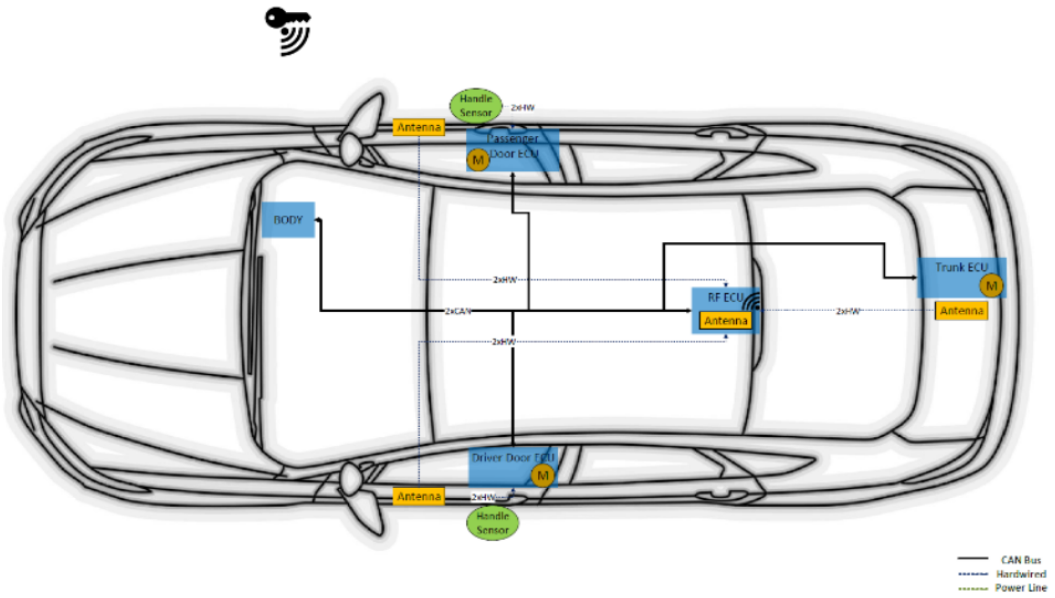


Figure 3.2: Vehicle architecture

The starting point of the process is the formal definition of system requirements within Cameo Systems Modeler, the MBSE tool adopted for this project. Requirements were systematically translated into functional models of the subsystems under analysis. To ensure clarity and completeness, both black-box views (describing the system in terms of external inputs and outputs) and white-box views (capturing the internal functional decomposition and potential allocations) were developed. This modeling activity provided a coherent and traceable representation of the system, ensuring that stakeholder needs, system functions, and architectural choices were formally documented and linked.
From this foundation, the integration between MBSE and MDO was realized by following a structured digital continuity workflow, as represented in Figure 3.3, that allowed information to flow seamlessly between the two environments. The process can be described in four major steps:

- Creation of the MBSE model in Cameo: requirements, functional architectures, and candidate allocations were defined within SysML diagrams, supported by tables to capture the relationships among system elements.

- Export of a functional model in matrix form: the information from Cameo was transformed into a CSV file that encapsulated functions, their interdependencies, and their weight. This functional matrix served as the structured input for the optimization phase, enabling the translation of descriptive system models into a quantitative problem formulation.
- The CSV dataset was imported into GEMSEO to define the optimization problem, which pursued the dual objective of minimizing both the total cost and the overall weight of the ECU network, while ensuring compliance with all system requirements. Parameters of the candidate ECUs, such as cost and mass, were sourced from an external database to maintain the realism and consistency of the optimization. Within this framework, GEMSEO implemented the selected optimization architecture and algorithm, systematically exploring alternative allocations of functions across ECU candidates and assessing the resulting trade-offs.
- Reintegration of results into the MBSE model: the optimized ECU configurations produced by GEMSEO were reintroduced into the Cameo model. This final step ensured that the optimized results were not treated as isolated outcomes of the optimization process but were fully embedded in the system model, preserving traceability to the original requirements and maintaining consistency across the life cycle.

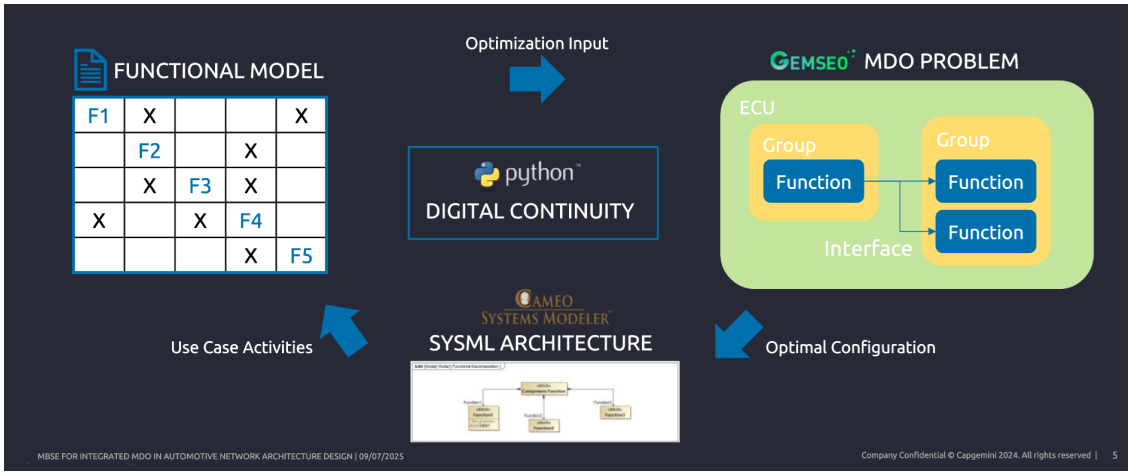


Figure 3.3: MBSE and MDO workflow

This iterative loop model creation, functional export, optimization, and reintegration constitutes the backbone of digital continuity between MBSE and MDO. It ensures that descriptive system models and quantitative optimization do not exist in separate silos but are connected within a coherent workflow. The advantage of this approach is twofold. On the one hand, MBSE guarantees the semantic integrity and traceability of requirements, functions, and architectures. On the other, MDO provides the computational capability to explore the design space, evaluate trade-offs, and propose quantitatively superior solutions.

The outputs of this process, as presented in the thesis, are therefore optimized ECU configurations and network that strike the best balance between cost and weight andsatisfies the design constraints. The case study thus demonstrates the potential of combining MBSE and MDO to address one of the most pressing challenges in automotive engineering: how to design electronic architectures that are not only functionally complete but also cost-efficient, lightweight, and systematically validated across the development life cycle.

## 3.3 From Stakeholder Needs to System Modeling: The Case Study Framework

### 3.3.1 System Modeling Approach

The model developed in Cameo Systems Modeler was structured according to the MagicGrid framework, which provides a systematic way to organize system information across multiple perspectives and levels of abstraction. The structure of the model reflects the natural progression from high-level stakeholder needs to detailed subsystem analysis, ensuring both completeness and traceability throughout the process.(3.4)

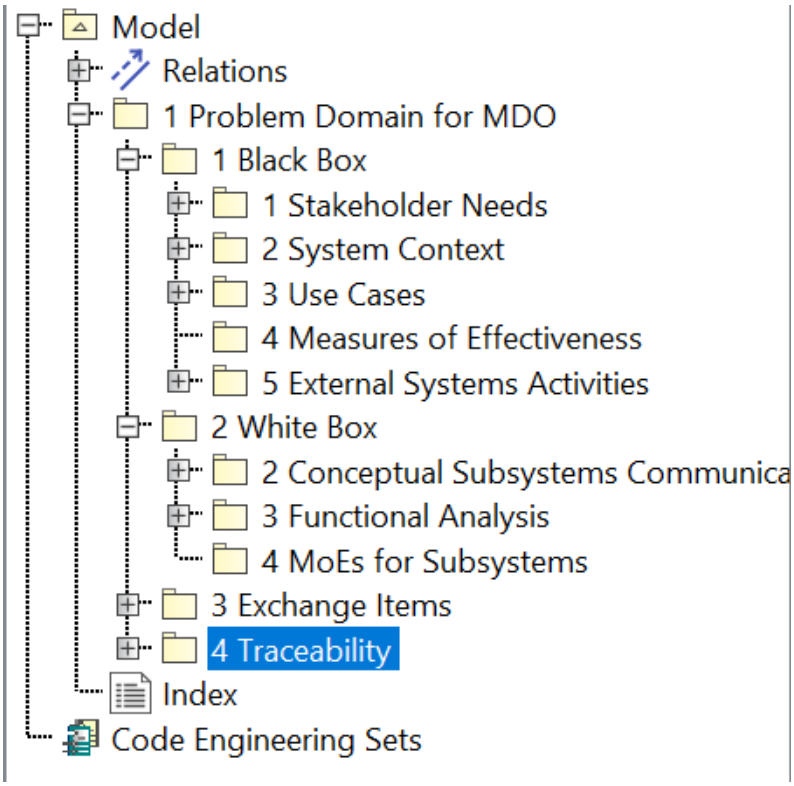


Figure 3.4: Model Structure

At the highest level, the Problem Domain for MDO was defined, serving as the root package of the case study. This domain was divided into several main sections:

- **Black Box**: This section captures the external view of the system. It begins with the identification of stakeholder needs, followed by the definition of the system context and the specification of relevant use cases. Additional elements such as measures of effectiveness (MoEs) and external system activities were included to provide a comprehensive description of the system environment and to establish criteria for assessing system performance.
- **White Box**: The white-box perspective provides the internal view of the system, focusing on its internal structure and behavior. This part of the model contains the conceptual subsystem communication architecture, which shows how subsystems interact, as well as the functional analysis, which decomposes high-level functions into more detailed ones. It also includes measures of effectiveness for subsystems, enabling the evaluation of performance at a finer granularity.
- **Exchange Items**: This section defines the flows of information and material exchanged among subsystems. Exchange items are essential for understanding the system's internal interactions and ensuring consistency between functional decomposition and subsystem communication.
- **Traceability**: To guarantee consistency and alignment across all levels of abstraction, a dedicated traceability package was created. This section establishes explicit links between stakeholder needs, system requirements, functions, subsystems, and exchange items. Traceability ensures that every design choice can be linked back to the original requirements, thus supporting verification and validation activities and enabling digital continuity with the optimization framework.

This structured approach to modeling offers several advantages. By distinguishing between external and internal perspectives, the model provides a holistic view of the system that captures both what the system must achieve and how it will achieve it. The layered representation improves clarity and reduces ambiguity, while the explicit definition of exchanges enhances understanding of interactions between subsystems. Most importantly, the traceability view guarantees consistency across all modeling levels, enabling designers to maintain alignment between requirements, functions, and architectural choices throughout the system development process. It is important to note, however, that in this case study the model represents a simplified version of a complete system model. In particular, the measures of effectiveness were not included, as the focus was placed primarily on capturing stakeholder needs, requirements, and functional interactions rather than on quantitative performance assessment.

### 3.3.2 Stakeholder Needs

The first step in the system definition process is the identification and formalization of stakeholder needs. Stakeholders are individuals or organizations that have a legitimate interest in the system, either because they interact directly with it, rely on its performance, or are responsible for its compliance with external standards. In the automotive context, stakeholders may include end-users, manufacturers, suppliers, regulatory authorities, and service providers, each with specific expectations regarding the behavior and

performance of the system.
Stakeholder needs describe what the system is expected to achieve, they capture the key functions and services that must be delivered to meet stakeholder objectives and therefore serve as the starting point for the entire requirements engineering process. From these needs, system requirements are derived and progressively refined into functional architectures and design solutions. In this case study, the stakeholder needs were identified by selecting the most significant functions considered essential for the system under analysis, drawing on prior company experience and previously executed projects, shown in Figure 3.5.

| # | Name | Text |
|---|---|---|
| 1 | SN-1.1 Lock the doors by pushing TLC B button | User shall be able to lock the doors by pushing TLC B button |
| 2 | SN-1.2 Unlock the doors by pushing TLC A button | User shall be able to unlock the doors by pushing TLC A button |
| 3 | SN-1.3 Doors Handle Pull with TLC within 1.5 m distance | Any doors shall open by pulling the handle with TLC within 1.5 m distance |
| 4 | SN-1.4 Lock\Unlock the trunk by pushing TLC T button | User shall be able to unlock\lock only the trunk by pushing TLC T button |
| 5 | SN-1.5 Constantly informed of locked/unlocked status of the vehicle | User shall be constantly informed about the overall locked/unlocked status of the vehicle |
| 6 | SN-1.6 Lock/Unlock by pushing Internal Button | User will be able to lock/unlock the vehicle using a button internal to the vehicle |
| 7 | SN-1.7 Light signals to inform about locking/unlocking operations | Vehicle shall have light signals to inform about locking/unlocking operations |
| 8 | SN-1.8 Unlock and Open if Trunk is close by pushing TLC T button | When the user push T button on TLC, if the Trunk is closed, the vehicle shall unlock and open the trunk |
| 9 | SN-1.9 Lock and Close if Trunk is open by pushing TLC T button | When the user push T button on TLC, if the Trunk is open, the vehicle shall close and lock the trunk |
| 10 | SN-1.10 Feedback on doors open/close status | The user shall be able to receive information on doors open/close status |
| 11 | SN-1.11 Feedback on trunk open/close status | The user shall be able to receive information on trunk open/close status |
| 12 | SN-1.12 Key Not inside the vehicle | The vehicle must ensure that the key does not remain locked inside the vehicle |
| 13 | SN-1.13 User lock from INT | The user shall be able to lock/unlock himself inside the vehicle |
| 14 | SN-1.14 User get out from vehicle | The user shall always be able to get out from the vehicle |
| 15 | SN-1.15 User set desired temperature | The system shall allow the user to manually set and adjust the desired cabin temperature |
| 16 | SN-1.16 Open/close windows | User shall be able to open/close windows from the inside of vehicle |

Figure 3.5: Stakeholder needs

The stakeholder needs can be grouped into three main categories: vehicle lock and unlock, the operation of doors and the trunk; window opening and closing; and climate control functions. Each needs is characterized by a unique identifier, a descriptive name, and a textual specification as shown in Figure 3.5.

- A first group of needs defines how users must be able to secure and access the vehicle using both the remote key fob (TLC) and in-vehicle controls. From the remote side, pressing the dedicated buttons must allow the user to lock the doors (TLC “B” button) and unlock them (TLC “A” button). Proximity-based access is required as well: when the user pulls the door handle while the TLC is detected within 1.5 m, the door must open, enabling a passive-entry interaction. Trunk access is also controlled by the TLC “T” button with state-dependent behavior: if the trunk is closed, pressing “T” shall unlock and open it; if the trunk is open, pressing “T” shall close and lock it. Complementing the remote functions, an internal lock/unlock button must provide equivalent capability from inside the cabin. This cluster also embeds status awareness and feedback: the user must be constantly informed of the vehicle’s locked/unlocked state, with light signals indicating lock/unlock operations and explicit feedback on both door and trunk open/close status. Two safety oriented needs further qualify these interactions: the key must not remain locked inside the vehicle, and egress must always be possible the user shall be able to get out of the vehicle regardless of prior lock commands. Together, these needs establish a

complete access-control experience: multiple actuation paths (remote and internal), context-aware trunk behavior, continuous HMI feedback, and safeguards to prevent lock-in or loss of access.

- A second group concerns window operation. Users must be able to open and close the windows from inside the vehicle. Although concise, this need sets a functional baseline that will drive the allocation of window-lift functions, the definition of related HMI elements, and the specification of interlocks (e.g., behavior when doors are locked or child-safety features in later refinements).

- The third group addresses climate control. The system must allow the user to manually set and adjust the desired cabin temperature. This establishes the core capability for HVAC (Heating, Ventilation, and Air Conditioning): a target-temperature setpoint managed through the climate interface, the system shall be able to reach the desired temperature set by the user.

### 3.3.3 System Context

The system context provides a high-level view of the actors involved in the interactions with the system and the types of information exchanged between them. In this case study, three primary entities are identified: the user, representing the driver or passenger of the vehicle; the vehicle, which encompasses the electronic and mechanical systems under analysis; and the TLC (key fob), which enables remote communication with the vehicle. Based on these elements, two distinct system contexts are defined: vehicle access and vehicle comfort. The vehicle access context shown in figure 3.6 describes the interactions related to locking, unlocking, and accessing the vehicle. In this context, three actors are present: the user, the vehicle, and the TLC. Several types of information flows are exchanged among them. From the user to the vehicle, the main signals are the handle pull and the external key signal, which trigger lock/unlock actions. From the vehicle to the user, multiple forms of feedback are provided, including internal and external lock/unlock status changes, external light signals, and open/close status changes for both doors and trunk. Between the TLC and the vehicle, dedicated signals ensure remote control and authentication, specifically the TLC command signals and the TLC search process that verifies proximity and authorization.

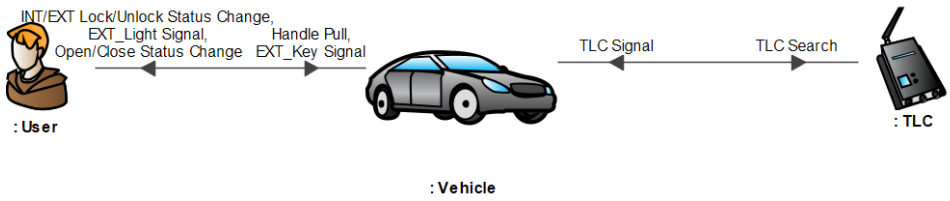


Figure 3.6: Vehicle access context

The vehicle comfort context focuses shown in figure 3.7 on functions associated with

passenger convenience, namely air conditioning and window control. In this case, the context includes only two actors: the user and the vehicle. The user provides inputs in the form of desired temperature settings, air conditioning (AC) button commands, and window button commands. In response, the vehicle returns information such as window open/close feedback and cabin temperature data, ensuring that the user is informed about the state of comfort-related functions.

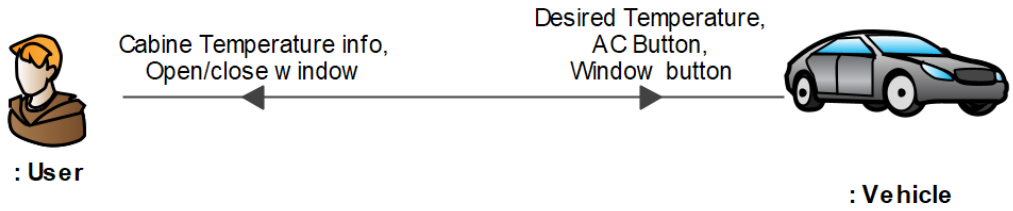


Figure 3.7: Vehicle comfort context

### 3.3.4 Use Cases

Use cases are a standard way of capturing the functional behavior of a system in systems engineering but also organizing that behavior. A use case describes the interactions that take place between the system being examined and actors external to the system (people, devices and/or other systems, etc.) in order to accomplish a goal. Use cases generally represent the interactions as a sequence of exchanging actions between the system and the actor and give an intuitive representation of how the system would be expected to behave in a real-world circumstance. The main advantage of use cases is that they provide a link between stakeholder needs, and system requirements. Use cases define the high-level purpose or goal for a system, and then enumerate the specific situations, establishing clarification not just as to what the system has to do, but also the conditions under which it must be done, and the type and kinds of interactions that occur. In this way use cases establish the boundaries of the system, establishing explicitly who or what will initiate an interaction with the system, what inputs the actors provide in relation to the interaction, and what outputs or response the system provides.
First, they improve communication across multidisciplinary stakeholders by providing a shared visual and textual language that avoids technical ambiguity. Second, they support traceability, as each use case can be linked directly to stakeholder needs, requirements, and subsequently to test cases.By being represented in modeling languages such as UML or SysML, they become part of a consistent model that integrates functional, structural, and behavioral views of the system. This allows use cases not only to act as high-level narratives, but also as formal modeling elements that can be connected to activities,

functions, and system components, thus ensuring a continuous and model-based representation of requirements and system behavior.
Building on the general principles described above, the use cases developed in this thesis have been organized into two main groups that reflect the functional domains of the system. The first group figure 3.8 concerns vehicle access, which includes all interactions related to locking, unlocking, and trunk operations. These scenarios involve both the user and the TLC (key fob) as actors and describe situations such as manual or remote door locking, unlocking and opening the trunk, passive entry, and safe vehicle exit. The second group figure3.9 relates to vehicle comfort, capturing functions that enhance the user's experience within the cabin, specifically the operation of windows and the adjustment of the cabin temperature through the HVAC system.

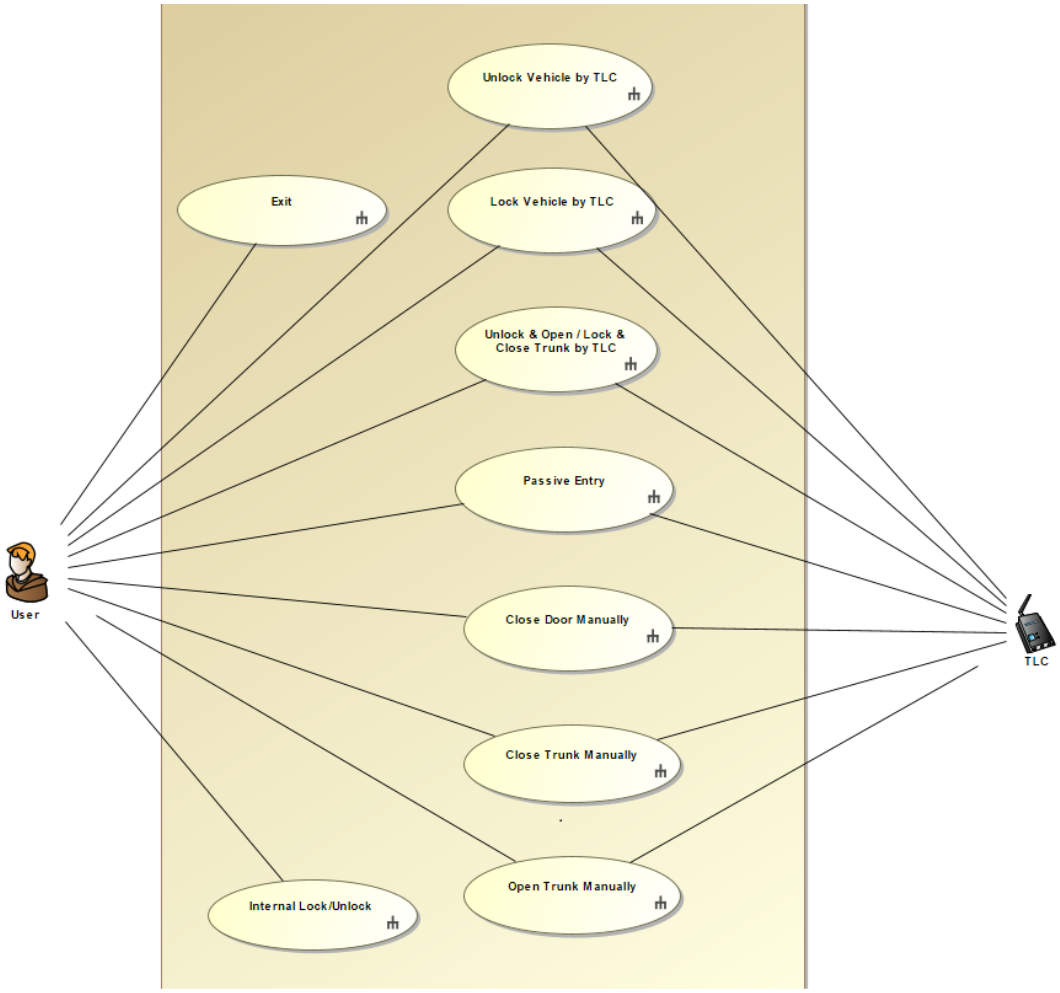


Figure 3.8: Use case: vehicle access

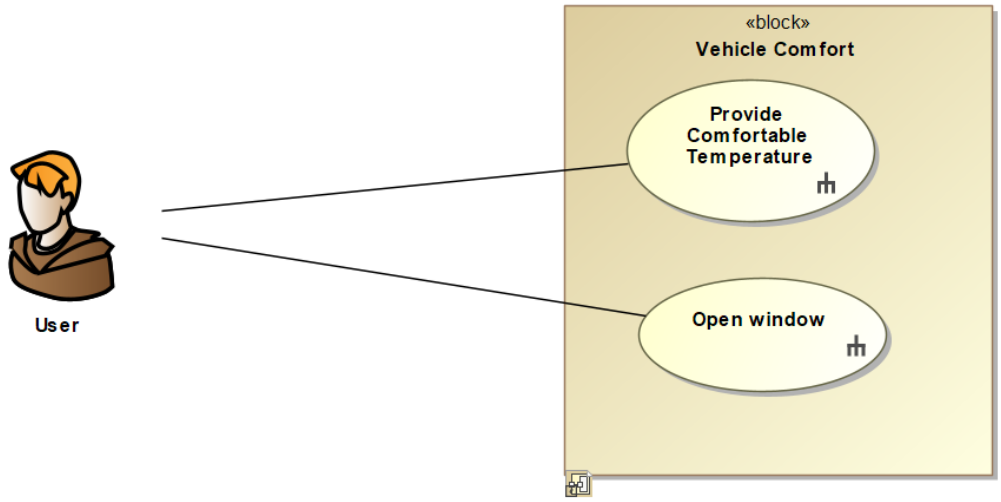


Figure 3.9: Use case: comfort

It should be noted that the case study represents a simplified version of a real automotive system, and therefore only a limited number of use cases were included. Nevertheless, these cases are sufficient to capture the essential functionalities and to illustrate the integration of MBSE practices with system optimization. Each one is refined into an activity diagram that details the sequence of actions, decision points, and interactions between the system and its actors. This ensures that the logic of every functional scenario is fully specified and provides a robust foundation for subsequent functional decomposition and allocation.
Through this structured approach, the use cases not only serve as a direct translation of stakeholder needs into operational scenarios, but also provide a clear, traceable link to the functional and architectural models developed later in the study.

### 3.3.5 Subsystem Structure

The vehicle system has been conceptually divided into a set of subsystems, each representing a specific functional domain: Sensors, Body Control, Ventilation System, Infotainment, Actuators, Entry Control, and the RF Transceiver. This decomposition enables a clearer representation of responsibilities and provides a structured view of how information flows between different parts of the vehicle.
As shown in the first block definition diagram figure 3.10, the model highlights the interfaces through which the subsystems communicate, making explicit all the signals exchanged among them. In particular, three high-level communication channels frame the

architecture: User Command, which represents the inputs generated by the user through handles, buttons, or other interactions; User Feedback, which provides information back to the user in the form of status changes, light signals, or notifications; and TLC Communication, which manages the interaction between the vehicle and the key fob.

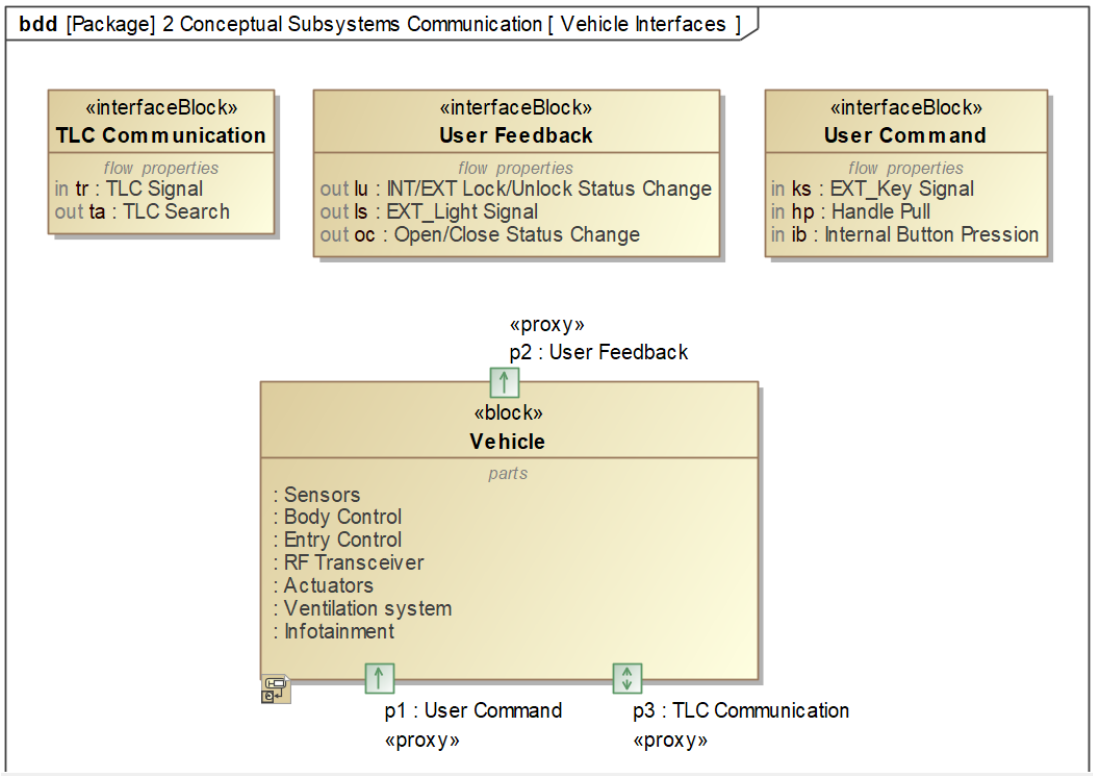


Figure 3.10: BDD: Vehicle interface

In addition to this top-level representation, a dedicated BDD was developed for each subsystem of the vehicle, the Body Control subsystem is shown in figure 3.11, which acts as the core coordinator of the vehicle architecture. This diagram specifies all the inputs and outputs associated with Body Control, including signals from sensors (e.g., vehicle status, trunk status, key location), commands to actuators (e.g., window operations, ventilation actuation), and feedback channels to the user. By explicitly modeling these interfaces, the Body Control unit is clearly represented as the central node that integrates information from external actors, processes it, and distributes the appropriate commands to other subsystems. These connections are further detailed through the use of interface blocks, which, when linked as inputs or outputs, explicitly capture all the signals exchanged with the Body Control. In this way, the diagram not only identifies the subsystems involved but also provides a precise specification of the information flows that ensure correct system operation. In figure 3.12 and 3.12 are shown other example of BDD of subsystems of the vehicle.

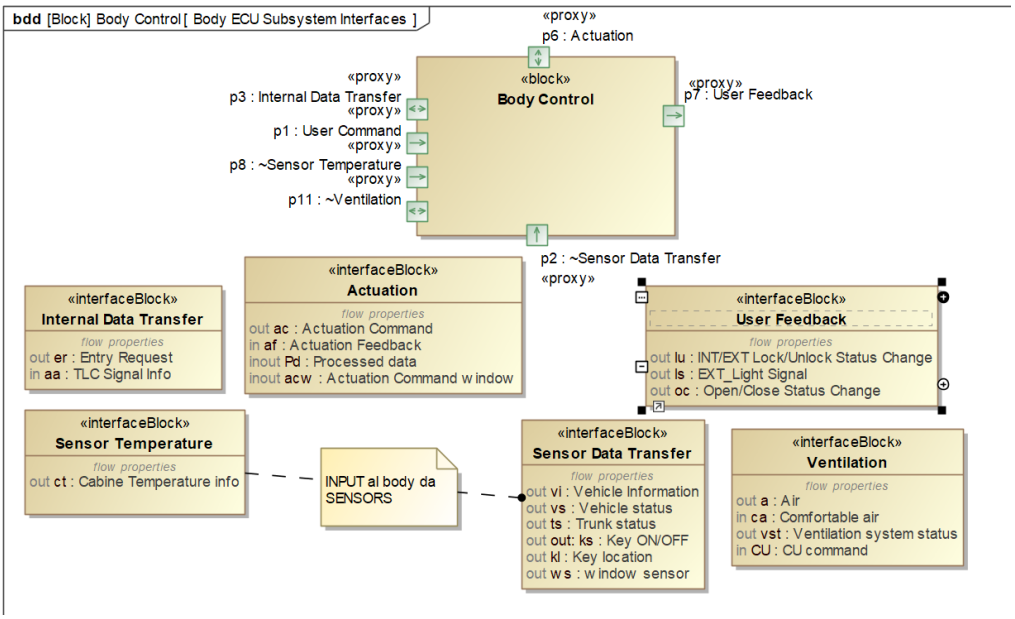


Figure 3.11: BDD: Body control

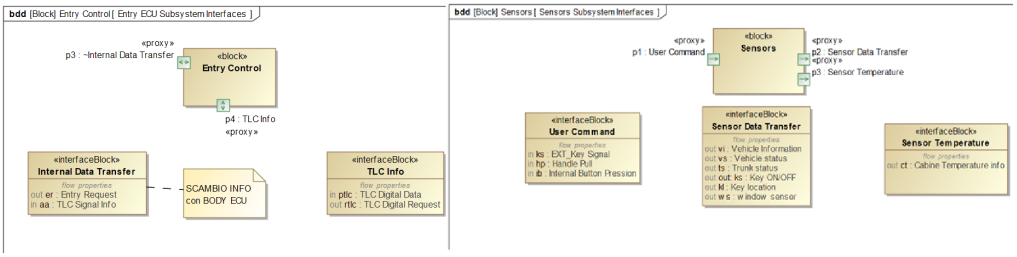


Figure 3.12: BDD: Body control and Sensor

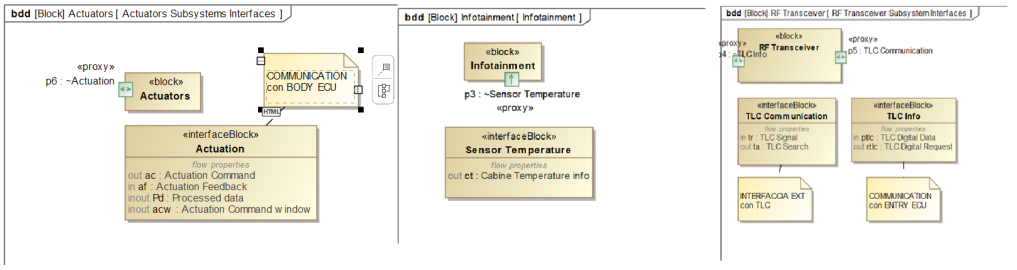


Figure 3.13: BDD: Actuator, Infotainment and RF Transciver

The Internal Block Diagram is used to describe the internal structure of a system block and the way its parts interact. Unlike the Block Definition Diagram, which provides a hierarchical view of the system's composition, the IBD focuses on the flow of information, energy, or material between the system's internal components. Its primary purpose is to represent the interfaces and connections among subsystems, making explicit how data and signals are exchanged.
In the IBD shown in figure 3.14, it is possible to observe both the external exchanges between the vehicle and its environment—represented by User Command, TLC Communication, and User Feedback, modeled as green pins placed on the diagram borders.For the internal interactions among the vehicle's subsystems in the diagram the blocks correspond to the individual subsystems, while the connecting arrows represent the links through which signals are exchanged. In this way, the diagram provides a clear visualization of how the vehicle integrates external inputs with internal processes, ensuring coherent communication across all functional domains.

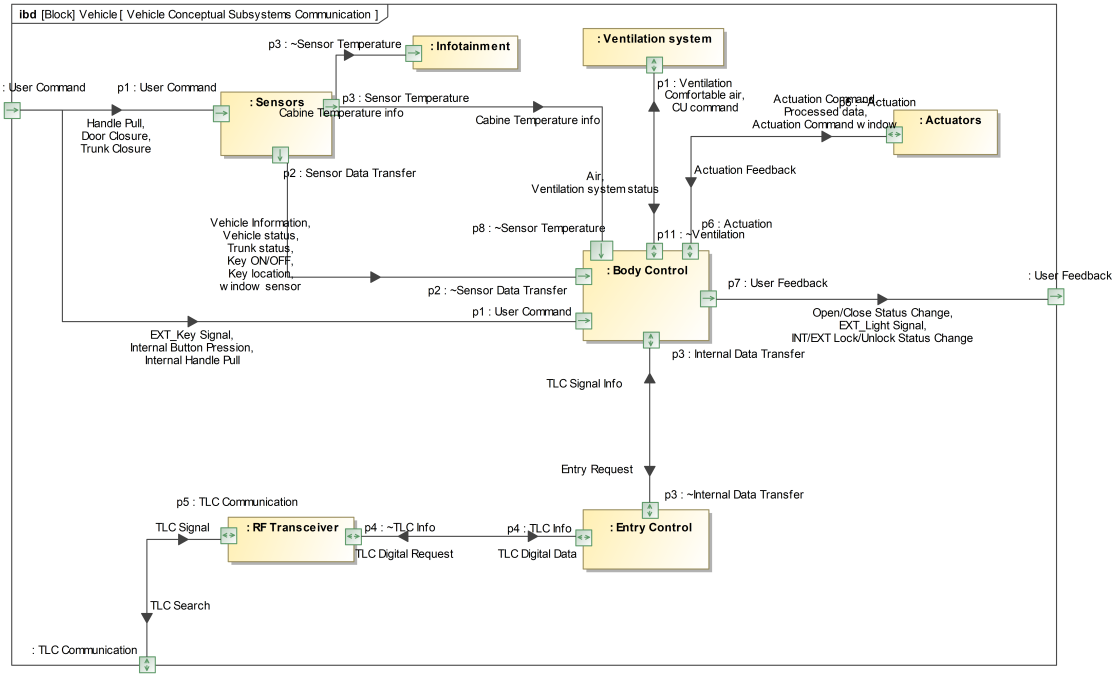


Figure 3.14: IBD: Vehicle

### 3.3.6 Exchange Items

In complex automotive systems, the definition and management of exchange items namely the signals exchanged among subsystems and between the system and its external environment represent a crucial step in the modeling process. These signals embody the flow of information that enables the execution of functions and the coordination of operations across the vehicle's architecture. By explicitly modeling them, it becomes possible to trace how stakeholder needs and system requirements are concretely implemented through communication among subsystems. To improve clarity and avoid confusion in system modeling, the exchange items identified in this case study have been categorized into two main groups:

- **Internal signals**, representing information exchanged within the vehicle architecture. These include communications among subsystems such as sensors, actuators, body control, ventilation, and entry control. Examples are the transfer of cabin temperature data, ventilation system status, vehicle information, all the internal signal are shown in figure 3.16
- **External signals**, representing interactions between the vehicle and external actors such as the user or the remote key (TLC), all the button, external feedback for the user ecc... All this signal are shown in figure 3.15 with thei classification.

This distinction, also represented in the figures, makes it easier to navigate the complexity of the system and supports traceability from requirements to implementation. Furthermore, it highlights the dual nature of automotive system interactions: on the one hand, the internal coordination among subsystems required to maintain consistency and performance, and on the other, the communication with the user that ensures correct and safe operation of vehicle functions.
The categorization of exchange items is not merely a descriptive exercise; it has practical implications for design, validation, and optimization. Grouping signals by their origin and destination clarifies responsibilities within the architecture, helps in identifying potential bottlenecks or redundancies, and facilitates the definition of verification and validation strategies tailored to specific categories of interaction.

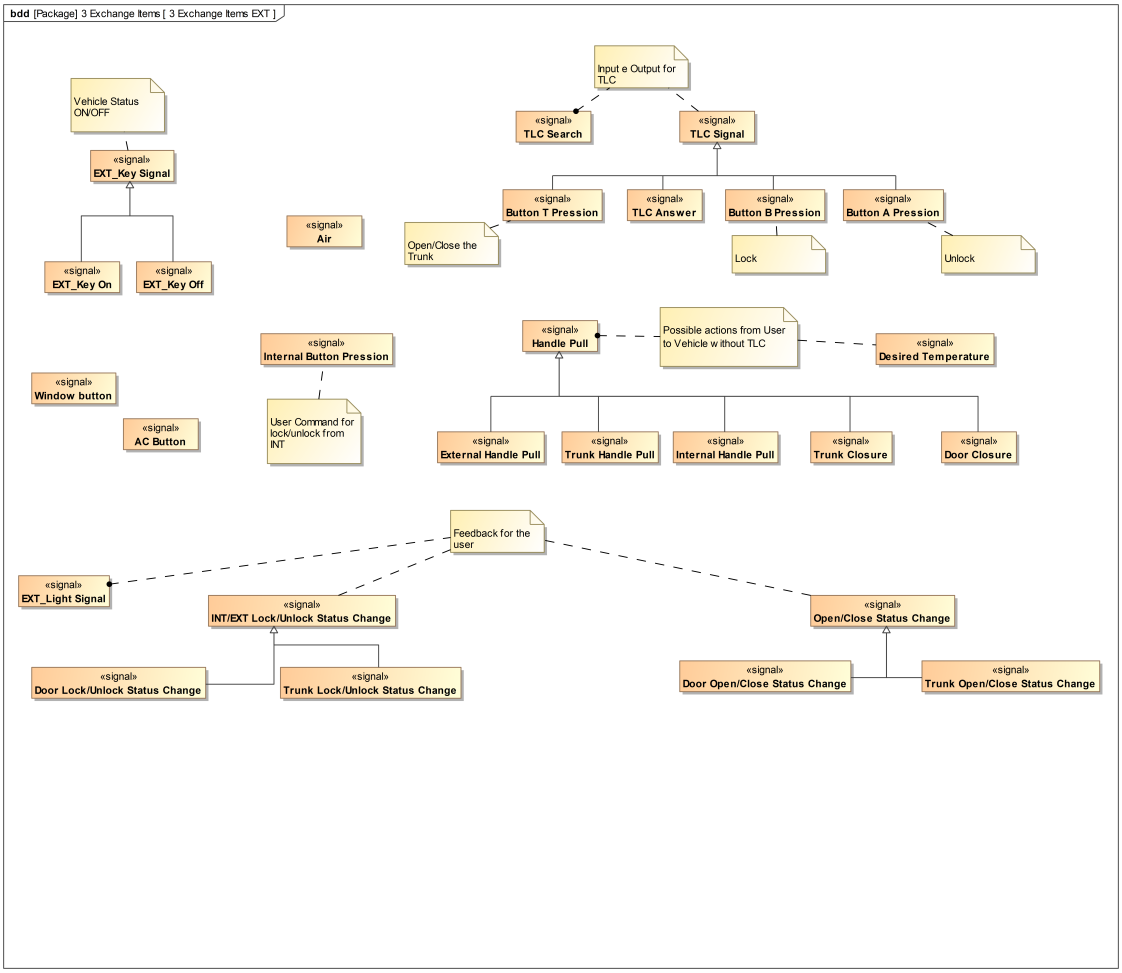


Figure 3.15: BDD: Exchange external items

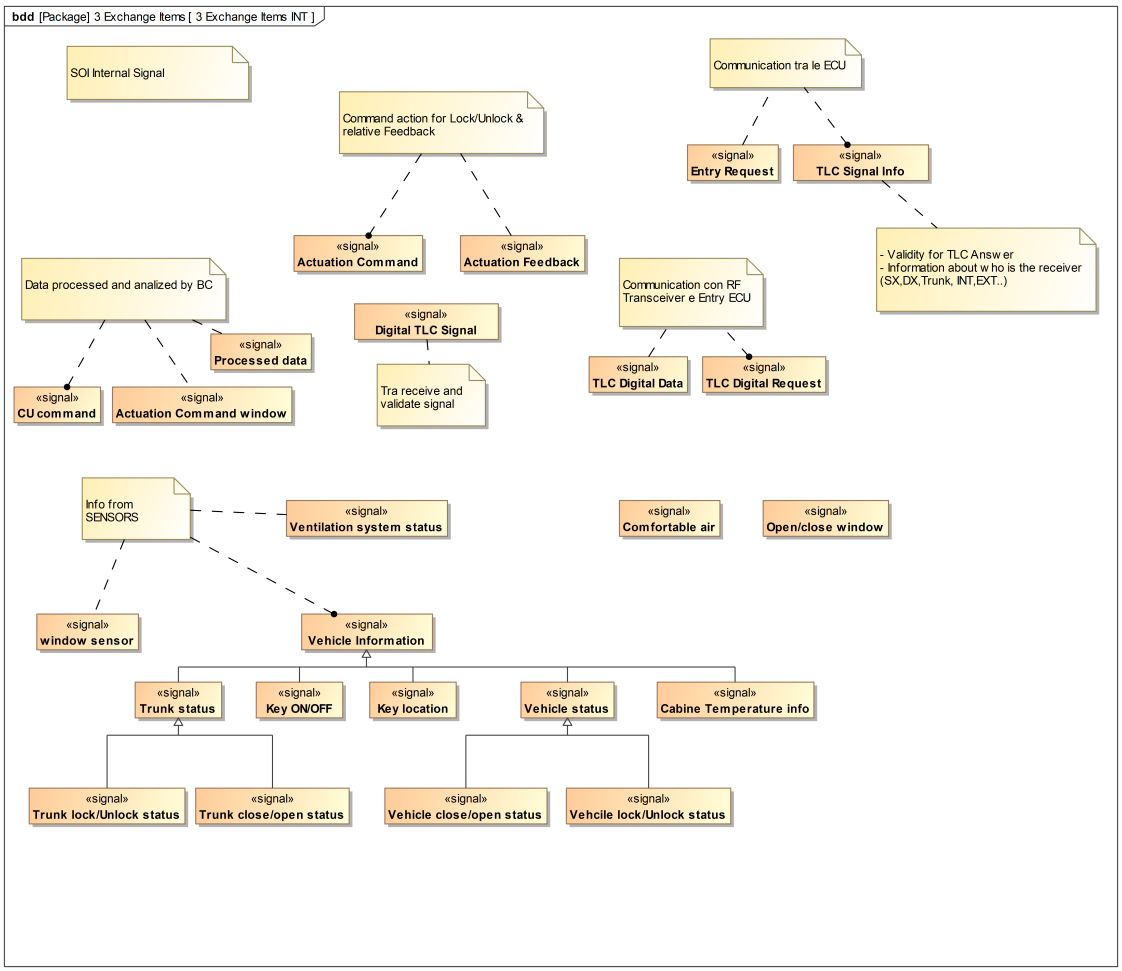


Figure 3.16: BDD: Exchange internal items

## 3.4 Activity diagram and Data Export Workflow

As previously discussed, each use case in the model is associated with a dedicated activity diagram, which provides a detailed view of how the corresponding functionality is executed. Activity diagrams are a key element of SysML modeling, as they describe the dynamic behavior of the system, illustrating the sequence of actions, decision points, and data flows that occur when a specific use case is performed. In this way, they bridge the gap between the high-level description of stakeholder interactions and the operational logic required to implement them.
In this project, several activity diagrams have been developed to capture the behavior of the different use cases identified earlier, such as vehicle access, trunk operations, and user feedback. However, in order to maintain clarity and avoid unnecessary redundancy, only two representative use cases will be presented in detail shown in Figure 3.8 and Figure 3.9. The first selected is the **Provide Comfortable Temperature** use case, which belongs to the broader category of Vehicle Comfort, the second one is the **Unlock vehicle by TLC**. This examples has been chosen because it encapsulates the interaction between user commands, subsystem coordination, and system feedback, making it a particularly illustrative case of the modeling approach adopted in this study.
In addition to the illustration of the activity diagrams, this part of the thesis will be explained how the system model can be systematically transformed into tables that serve as a bridge between MBSE and MDO. These tables, extracted directly from the model are formatted to be used as input for the optimization framework implemented in Python.

### 3.4.1 Provide Comfortable Temperature

The *"Provide Comfortable Temperature"* use case is modeled with two swimlanes User and Vehicle so that the allocation of responsibilities is explicit: actions in the User lane represent external interactions with the system, while actions in the Vehicle lane capture the internal system behavior needed to realize the function.
At a high level, the sequence proceeds as follows, figure 3.17 The User initiates the service by turning on the climate control. The Vehicle then checks the system and, if operational, starts climate control. The vehicle measures the cabin temperature and displays this information to the user. The User then evaluates comfort; if the current state is not satisfactory, the user sets a desired temperature. The vehicle proceeds to provide air and iterates the control loop until the desired temperature is reached. Once the user is satisfied, or explicitly requests it, the system stops climate control, and the activity terminates. In figure 3.17 each Activity parameter node present an activity diagram the shows the flow of function inside the vehicle, it is useful to analize which subsystem does the function. As an example in figure 3.18 is shown The *"Reach Desired Temperature"* activity, it provides a detailed representation of the internal control logic that enables the vehicle to deliver the user's requested cabin temperature. The diagram is structured around two subsystems, Body Control and the Ventilation System, with clear inputs and outputs that define the interaction between them.
The process begins with the acquisition of three critical inputs: the Desired Temperature set by the user, the Cabin Temperature information measured by onboard sensors, and

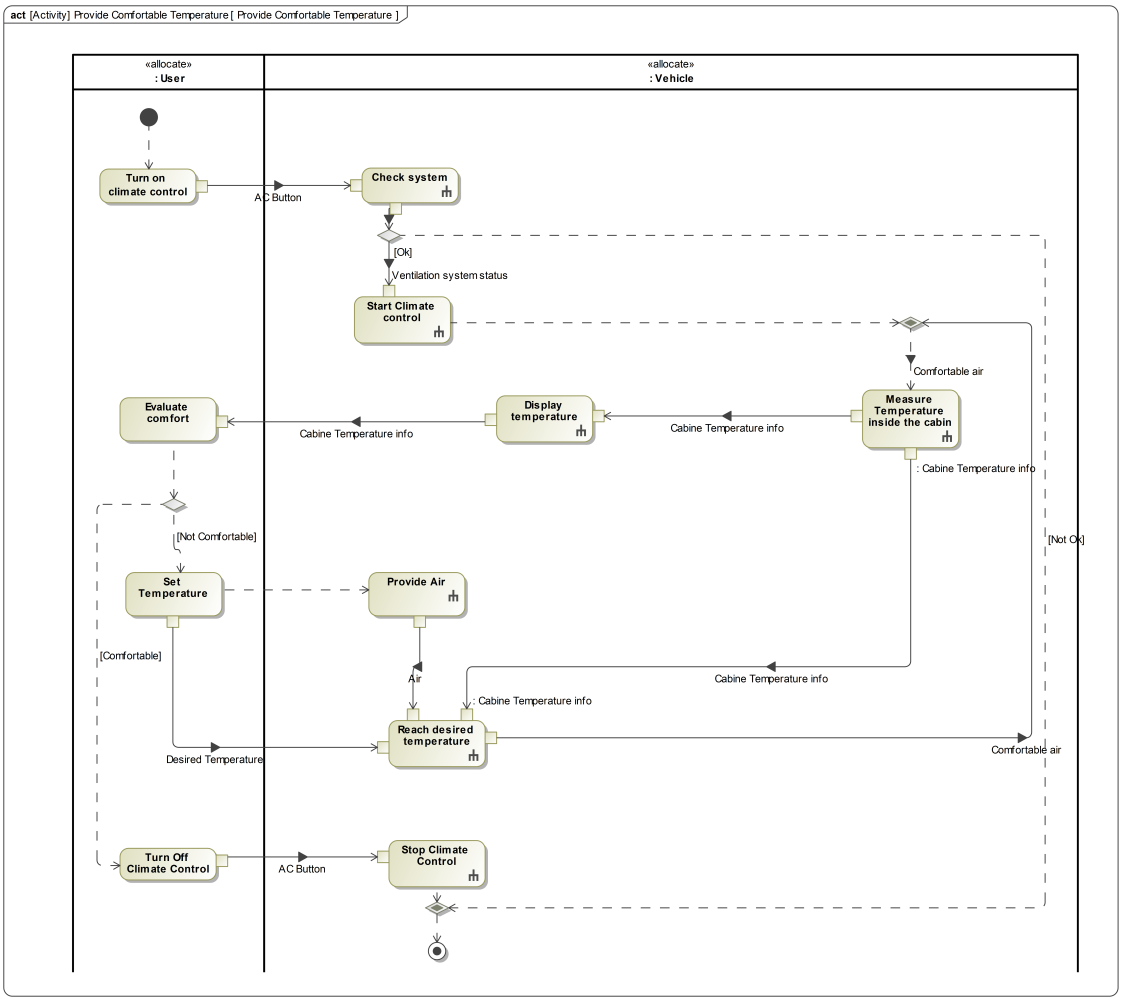


Figure 3.17: Activity diagram: Provide Comfortable Temperature activity

the Air available to the system. These inputs are first processed within the Ventilation System, where the cabin temperature is read and transferred to the computing unit for further evaluation. At the same time, the Body Control unit receives both the cabin data and the user's setpoint and undertakes a comparison between the actual and desired temperature.

Based on this assessment, the system follows one of two possible branches. If the cabin temperature is higher than desired, the Body Control initializes the cooling sequence; conversely, if the cabin temperature is lower than desired, the heating sequence is activated. In either case, Body Control generates a CU command that orchestrates the Ventilation System's actions.For cooling, the Ventilation System draws air out from the cabin, reduces its temperature to the required level, and blows the cooled air back into the interior. For heating, the process mirrors this logic: the system extracts air from the cabin, warms it to the setpoint, and reintroduces it into the cabin. These operations ensure a continuous flow of treated air, progressively steering the cabin environment toward

the desired conditions.

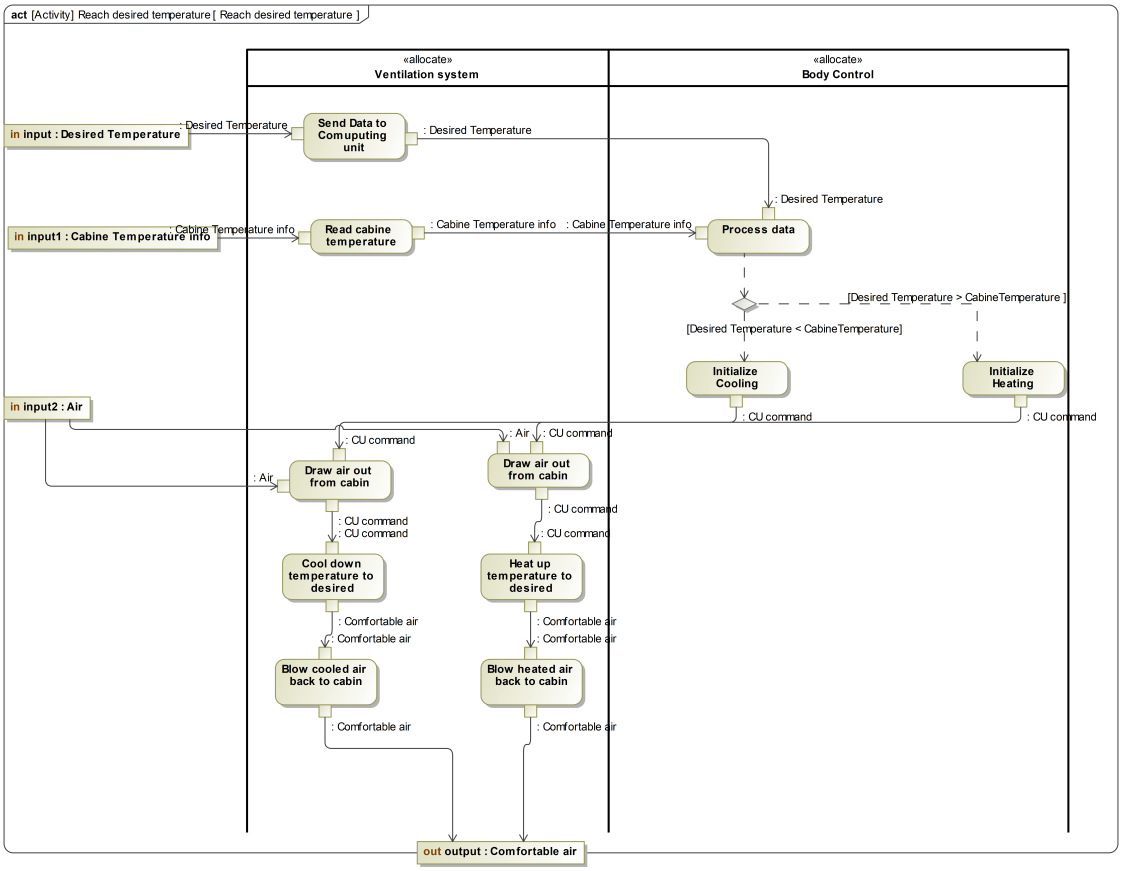


Figure 3.18: Activity diagram: Reach desired temperature activity

### 3.4.2 Unlock vehicle by TLC

The Unlock Vehicle by TLC activity diagram, figure 3.19 provides a structured representation of how the vehicle interacts with the user and the TLC device in order to execute the unlocking operation. The process is initiated when the user presses the unlocking button on the TLC (Send Unlock Command). This input triggers the TLC to generate a digital signal, which is subsequently received (Receive TLC Signal) and validated (Validate TLC Signal). Validation serves as a safeguard: if the received signal does not conform to the expected protocol or security criteria, the process is immediately aborted, thereby preventing unauthorized access.
If the validation is successful, the verified TLC Signal Info is transmitted to the vehicle subsystem, which undertakes the critical task of checking its current state (Check Vehicle Status). This step determines whether the vehicle is already unlocked or requires an unlocking command. Should the vehicle be locked, the control logic formulates and transmits an Actuation Command through the Send Command activity, which is elaborated by the Body Control unit. This command initiates the unlocking process, physically engaging the actuators and changing the state of the locking mechanism.

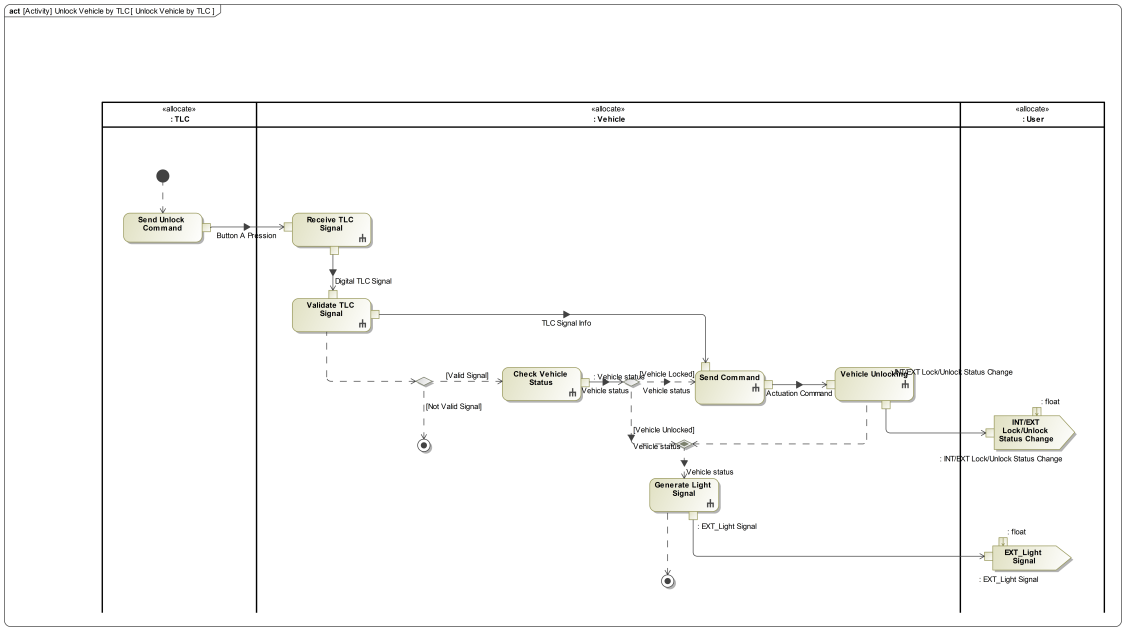


Figure 3.19: Activity diagram: Unlock Vehicle by TLC activity

Upon successful unlocking, feedback is generated at two distinct levels. First, the system updates internal and external status signals (INT/EXT Lock/Unlock Status Change),second, a user-oriented feedback mechanism is activated by generating an EXT Light Signal, thereby providing the driver with immediate confirmation of the completed action.
From a structural perspective, the diagram is organized into swimlanes that clearly delineate responsibilities across the user, the TLC, and the vehicle. The user initiates the

process by interacting with the TLC, the TLC is responsible for generating, transmitting, and validating the unlocking command, and the vehicle processes validated signals, verifies conditions, and physically actuates the unlocking mechanism. This structured decomposition highlights not only the logical sequence of operations but also the importance of validation and feedback mechanisms.
As outlined in the previous section, each activity parameter node is associated with an activity diagram that allocates specific functions to the corresponding subsystems. In the case illustrated, the Send Command function is represented in figure 3.20. Unlike the more complex processes discussed earlier, this diagram highlights a single function allocated entirely to the Body Control subsystem. The activity is characterized by a straightforward structure, comprising one input (TLC Signal Info) and one output (Actuation Command). This simplicity stands in contrast to the multi-layered interactions of previous activity diagrams, underscoring how certain functions can be modeled with a reduced degree of complexity while still maintaining clarity in the representation of signal flows.

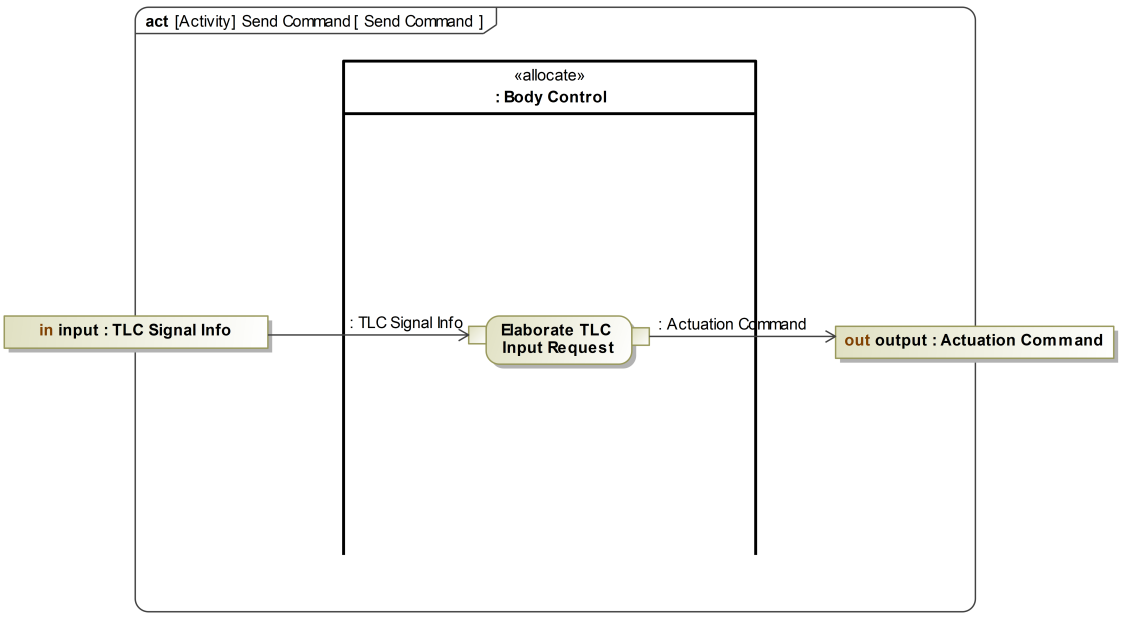


Figure 3.20: Send Command activity

### 3.4.3 Tabular Data Extraction and Structuring for MBSE–MDO Coupling

In order to establish a seamless connection between Model-Based Systems Engineering and Multidisciplinary Design Optimization, it is necessary to transform the information contained in the system model into a format that can be readily interpreted by external optimization tools. For this purpose, a series of tables has been developed. These tables act as an intermediate layer between the SysML-based system model and the Python/GEMSEO optimization environment, ensuring that functional, structural, and signal-related information is transferred in a consistent and machine-readable way.
The data extracted from the system model is exported in the form of CSV files. A CSV file is a widely used plain-text format in which data is organized in rows and columns, with each value separated by a delimiter (commonly a comma or semicolon). This structure makes CSV files both human-readable and easily processable by software tools.
To enable the transfer of data into Python for the optimization phase, three dedicated tables were created. The table 3.1 called **MDO input** is organized into three columns: the first specifies whether the element is classified as an input or an output, the second describes the function performed, and the third identifies the signal exchanged. The purpose of this table is to provide a comprehensive mapping of all system functions, clearly associating them with their corresponding inputs and outputs, and thereby making explicit the relationships that govern their interactions. In addition, the table highlights cases in which two functions share the same type: if one function produces an output and another consumes it as an input, the signal is thereby identified as being exchanged between the two functions.

| **I/O signal** | **Owner** | **Type** |
|---|---|---|
| input | :Terminate Climate control | AC Button |
| input | :Check ventilation system status | AC Button |
| input | :Lock Door & Trunk from INT | Actuation Command |
| input | :Unlock Trunk | Actuation Command |
| input | :Close Trunk | Actuation Command |
| input | :Lock Trunk | Actuation Command |
| input | :Unlock Door & Trunk | Actuation Command |
| input | :Lock Door & Trunk | Actuation Command |
| input | :Open Trunk | Actuation Command |
| input | :Lock Door | Actuation Command |
| input | :Open or Close window | Actuation Command window |
| input | :elaborate request | Actuation Command window |
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Generate Vehicle Open/Close External Feedback | Actuation Feedback |

| **I/O signal** | **Owner** | **Type** |
|---|---|---|
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Generate Vehicle Open/Close External Feedback | Actuation Feedback |
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Generate Vehicle Lock/Unlock External Feedback | Actuation Feedback |
| input | :Draw air out from cabin | Air |
| input | :Draw air out from cabin | Air |
| input | :Display Temperature | Cabine Temperature info |
| input | :Read cabine temperature | Cabine Temperature info |
| input | :Blow heated air back to cabin | Comfortable air |
| input | :Blow cooled air back to cabin | Comfortable air |
| input | :Heat up temperature to desired | CU command |
| input | :Cool down temperature to desired | CU command |
| input | :Send Data to Comuputing unit | Desired Temperature |
| input | :Process data | Desired Temperature |
| input | :Validity Analysis of TLC input | Digital TLC Signal |
| input | :Elaborate Search Signal | Entry Request |
| input | :Receive External Command | Handle Pull |
| input | :Receive External Command | Handle Pull |
| input | :Receive External Command | Handle Pull |
| input | :Receive External Command | Handle Pull |
| input | :Receive External Command | Handle Pull |
| input | :Receive Internal Lock/Unlock Button Press | Internal Button Pression |
| input | :Trasform processed data into actuator input | Processed data |
| input | :Send Search Signal | TLC Digital Request |
| input | :Convert RF TLC input to Digital | TLC Signal |
| input | :Elaborate TLC Input Request | TLC Signal Info |
| input | :Elaborate Key Position (INT or EXT) | TLC Signal Info |
| input | : Internal check of trunk status | Trunk lock/Unlock status |
| input | :Internal Check of Vehicle Status | Vehcile lock/Unlock status |
| input | :Generate Close/Open Info Feedback | Vehicle Information |
| input | :Communicate Command Reception | Vehicle Information |

| **I/O signal** | **Owner** | **Type** |
|---|---|---|
| input | :Generate Close/Open Info Feedback | Vehicle Information |
| input | :Generate Close/Open Info Feedback | Vehicle Information |
| input | :Generate Close/Open Info Feedback | Vehicle Information |
| input | :Inizialize climate control | Ventilation system status |
| input | :Verify status of window how much open or close | Window button |
| input | :Evaluate data and trasform it into an input | window sensor |
| input1 | :Process data | Cabine Temperature info |
| input1 | :Draw air out from cabin | CU command |
| input1 | :Draw air out from cabin | CU command |
| input1 | : Internal check of trunk status | Trunk close/open status |
| input1 | :Internal Check of Vehicle Status | Vehicle close/open status |
| output | :Elaborate Internal Command | Actuation Command |
| output | :Elaborate TLC Input Request | Actuation Command |
| output | :Elaborate Internal Command | Actuation Command |
| output | :Trasform processed data into actuator input | Actuation Command window |
| output | :elaborate request | Actuation Command window |
| output | :Lock Door & Trunk from INT | Actuation Feedback |
| output | :Unlock Door & Trunk | Actuation Feedback |
| output | :Open Trunk | Actuation Feedback |
| output | :Close Trunk | Actuation Feedback |
| output | :Lock Trunk | Actuation Feedback |
| output | :Unlock Trunk | Actuation Feedback |
| output | :Lock Door | Actuation Feedback |
| output | :Lock Door & Trunk | Actuation Feedback |
| output | :Read cabine temperature | Cabine Temperature info |
| output | :Measure Temperature inside the cabin | Cabine Temperature info |
| output | :Display Temperature | Cabine Temperature info |
| output | :Blow heated air back to cabin | Comfortable air |
| output | :Blow cooled air back to cabin | Comfortable air |
| output | :Cool down temperature to desired | Comfortable air |
| output | :Heat up temperature to desired | Comfortable air |
| output | :Initialize Cooling | CU command |
| output | :Draw air out from cabin | CU command |
| output | :Initialize Heating | CU command |
| output | :Draw air out from cabin | CU command |
| output | :Send Data to Comuputing unit | Desired Temperature |
| output | :Convert RF TLC input to Digital | Digital TLC Signal |
| output | :Communicate Command Reception | Entry Request |

| **I/O signal** | **Owner** | **Type** |
|---|---|---|
| output | :Generate Visual Feedback | EXT_Light Signal |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Generate Vehicle Lock/Unlock External Feedback | INT/EXT Lock/Unlock Status Change |
| output | :Elaborate Key Position (INT or EXT) | Key location |
| output | :Verify if Power is ON or OFF | Key ON/OFF |
| output | :Generate Close/Open Info Feedback | Open/Close Status Change |
| output | :Generate Vehicle Open/Close External Feedback | Open/Close Status Change |
| output | :Generate Close/Open Info Feedback | Open/Close Status Change |
| output | :Generate Close/Open Info Feedback | Open/Close Status Change |
| output | :Generate Close/Open Info Feedback | Open/Close Status Change |
| output | :Generate Vehicle Open/Close External Feedback | Open/Close Status Change |
| output | :Open or Close window | Open/close window |
| output | :Evaluate data and trasform it into an input | Processed data |
| output | :Elaborate Search Signal | TLC Digital Request |
| output | :Send Search Signal | TLC Search |
| output | :Validity Analysis of TLC input | TLC Signal Info |
| output | :Verify if Trunk is Open/Close | Trunk close/open status |
| output | :Verify if Trunk is Locked/Unlocked | Trunk lock/Unlock status |
| output | : Internal check of trunk status | Trunk status |
| output | :Verify if Vehicle is Locked/Unlocked from INT/EXT | Vehcile lock/Unlock status |
| output | :Verify if Vehicle is Close/Open | Vehicle close/open status |
| output | :Receive External Command | Vehicle Information |

| **I/O signal** | **Owner** | **Type** |
|---|---|---|
| output | :Receive External Command | Vehicle Information |
| output | :Receive External Command | Vehicle Information |
| output | :Receive External Command | Vehicle Information |
| output | :Receive External Command | Vehicle Information |
| output | :Internal Check of Vehicle Status | Vehicle status |
| output | :Check ventilation system status | Ventilation system status |
| output | :Verify status of window how much open or close | window sensor |
| output1 | :Suck air from outside | Air |
| output1 | :Measure Temperature inside the cabin | Cabine Temperature info |

Table 3.1: MDO input

The table 3.2, referred to as the **Function Weight** table, is designed to capture the computational load associated with each vehicle function. This table is structured into three columns. The first column of the table indicates the functional group to which each function belongs,function have been classified according to their purpose and similarity, in order to cluster them in the most consistent way. This categorization aims to support the allocation of similar tasks to the same control unit, thereby improving modularity and reducing implementation complexity.The second column lists all the functions that the ECUs must perform, to ensure clarity and consistency, only the functions previously identified within the activity diagrams and specifically allocated to the Vehicle swimlanes and its subsystems are included. The third column assigns a category to each function and the fourth a numerical value expressed in DMIPS (Dhrystone Million Instructions Per Second).
DMIPS is a widely used benchmark metric that measures the processing capability of a microprocessor by quantifying how many millions of Dhrystone instructions it can execute per second. The Dhrystone benchmark, originally developed as a synthetic performance test, has become an industry-standard reference point for evaluating embedded systems. In the context of ECUs, DMIPS provides a normalized way to assess and compare computational requirements across diverse hardware platforms.

| **Group** | **Function map:** | **Category** | **DMIPS** |
|---|---|---|---|
| 1 | Terminate Climate control | A | 3 |
| 1 | Check ventilation system status | C | 20 |
| 2 | Lock Door & Trunk from INT | A | 3 |
| 2 | Unlock Trunk | A | 3 |
| 2 | Close Trunk | A | 3 |
| 2 | Lock Trunk | A | 3 |
| 2 | Unlock Door & Trunk | A | 3 |
| 2 | Lock Door & Trunk | A | 3 |
| 2 | Open Trunk | A | 3 |
| 2 | Lock Door | A | 3 |
| 9 | Open or Close window | A | 3 |
| 3 | elaborate request | D | 30 |
| 4 | Generate Vehicle Lock/Unlock External Feedback | A | 3 |
| 4 | Generate Vehicle Open/Close External Feedback | A | 3 |
| 5 | Draw air out from cabin | B | 10 |
| 1 | Display Temperature | A | 3 |
| 1 | Read cabine temperature | A | 3 |
| 5 | Blow heated air back to cabin | B | 10 |
| 5 | Blow cooled air back to cabin | B | 10 |
| 1 | Heat up temperature to desired | C | 20 |
| 1 | Cool down temperature to desired | C | 20 |
| 1 | Send Data to Comuputing unit | C | 20 |
| 1 | Process data | C | 20 |

| Group | Function map | Category | DMIPS |
|---|---|---|---|
| 1 | Validity Analysis of TLC input | A | 3 |
| 1 | Elaborate Search Signal | A | 3 |
| 1 | Receive External Command | D | 30 |
| 1 | Receive Internal Lock/Unlock Button Press | D | 30 |
| 1 | Trasform processed data into actuator input | D | 30 |
| 1 | Send Search Signal | D | 30 |
| 1 | Convert RF TLC input to Digital | D | 30 |
| 1 | Elaborate TLC Input Request | D | 30 |
| 1 | Elaborate Key Position (INT or EXT) | D | 30 |
| 1 | Internal check of trunk status | D | 30 |
| 1 | Internal Check of Vehicle Status | D | 30 |
| 1 | Generate Close/Open Info Feedback | D | 30 |
| 1 | Communicate Command Reception | D | 30 |
| 1 | Inizialize climate control | D | 30 |
| 1 | Verify status of window how much open or close | D | 30 |
| 1 | Evaluate data and trasform it into an input | D | 30 |
| 5 | Process data | B | 10 |
| 5 | Draw air out from cabin | B | 10 |
| 5 | Internal check of trunk status | B | 10 |
| 5 | Internal Check of Vehicle Status | B | 10 |
| 5 | Generate Close/Open Info Feedback | B | 10 |
| 5 | Communicate Command Reception | B | 10 |
| 5 | Inizialize climate control | B | 10 |
| 5 | Verify status of window how much open or close | B | 10 |
| 5 | Evaluate data and trasform it into an input | B | 10 |
| 5 | Suck air from outside | B | 10 |

Table 3.2: Function Weigh

By associating each system function with a weight in DMIPS, the table enables a direct estimation of the processing demand imposed on the vehicle's electronic architecture. This approach allows system engineers to match functions with appropriate ECUs, ensuring that the available computational capacity is sufficient to guarantee reliable operation. Table 3.3 presents the classification of system functions into four categories, defined according to their level of complexity and computational demand, each category is described by a short definition and is associated with an estimated DMIPS value, which provides a quantitative indication of the processing effort required. The categories can be summarized as follows:

- **Category A**: Groups the most basic functions, limited to signal acquisition or output transmission without further elaboration, sometimes including simple visual feedback.
- **Category B**: Encompasses functions responsible for handling internal commands

and implementing control logic, ensuring proper coordination of subsystem operations.

- **Category C**: Refers to more complex functions that involve algorithmic processing, signal analysis, and interactions across multiple subsystems.
- **Category D**: Covers the most computationally intensive functions, characterized by advanced data elaboration and processing activities.

| Category | Description | DMIPS |
|---|---|---|
| A | Input Acquisition or Output sending without elaboration or Visual Feedback Signal | 3 |
| B | Internal Command Handling and Control Logic Signal | 10 |
| C | Complex Algorithm, Signal Analysis and Multiple System Interaction | 20 |
| D | Elaboration and procession of data | 30 |

Table 3.3: Category classification with description and DMIPS value

The table 3.4 is called **Functional Interface Weight** table, which focuses on the computational weight of the signals exchanged between subsystems and ECUs. Unlike the previous Function Weight table, which quantified the processing demand of functions, this table addresses the communication layer of the architecture. By assigning weights to the various signals, it becomes possible to evaluate and configure the data traffic exchanged over the in-vehicle communication buses.
This information is essential for determining the most appropriate allocation of signals to specific communication channels (e.g., CAN, LIN, or Ethernet) and for assessing the overall bus load under different configurations. In practice, the table enables system engineers to anticipate possible bottlenecks, balance communication demands across the architecture, and ensure that the design of the ECU network satisfies both performance and reliability requirements.

| Signal name | Computational load (DMIPS) |
|---|---|
| AC Button | 1 |
| Actuation Command | 2 |
| Actuation Command window | 2 |
| Actuation Feedback | 2 |
| Air | 1 |
| Button A Pression | 1 |
| Button B Pression | 1 |
| Button T Pression | 1 |
| Cabine Temperature info | 3 |
| Comfortable air | 1 |
| CU command | 2 |

| **Signal name** | **Computational load (DMIPS)** |
|---|---|
| Desired Temperature | 1 |
| Digital TLC Signal | 3 |
| Door Closure | 2 |
| Door Lock/Unlock Status Change | 3 |
| Door Open/Close Status Change | 3 |
| Entry Request | 3 |
| EXT_Key Off | 1 |
| EXT_Key On | 1 |
| EXT_Key Signal | 3 |
| EXT_Light Signal | 3 |
| External Handle Pull | 1 |
| Handle Pull | 1 |
| INT/EXT Lock/Unlock Status Change | 2 |
| Internal Button Pression | 1 |
| Internal Handle Pull | 1 |
| Key location | 3 |
| Key ON/OFF | 1 |
| Open/Close Status Change | 6 |
| Open/close window | 3 |
| Processed data | 6 |
| TLC Answer | 2 |
| TLC Digital Data | 1 |
| TLC Digital Request | 2 |
| TLC Search | 1 |
| TLC Signal | 1 |
| TLC Signal Info | 1 |
| Trunk close/open status | 2 |
| Trunk Closure | 1 |
| Trunk Handle Pull | 1 |
| Trunk lock/Unlock status | 2 |
| Trunk Lock/Unlock Status Change | 6 |
| Trunk Open/Close Status Change | 6 |
| Trunk status | 2 |
| Vehicle lock/Unlock status | 2 |
| Vehicle close/open status | 2 |
| Vehicle Information | 3 |
| Vehicle status | 3 |
| Ventilation system status | 3 |
| Window button | 1 |
| Window sensor | 3 |

Table 3.4: Functional Interface weight

As was done for the Function Weight table, the different signals have been categorized into predefined groups, and each category has been assigned an estimated computational weight. These values, expressed in DMIPS, were determined according to the information load associated with the exchanged signal. In this way, signals carrying more complex or data-intensive information are assigned higher DMIPS values, while simpler signals are associated with lower computational demands.
This categorization is explained in table 3.5.

| **Description** | **DMIPS** |
|---|---|
| Actuation feedback signals are very simple, often boolean, and confirm the status of actuators (e.g., locked/unlocked). These signals involve a straightforward reading of the actuator's state and transmitting this information. Processing is minimal. Feedback signals are smaller in content compared to commands and are generally boolean. | 1 |
| Basic and moderately complex commands involving the control of actuators or the management of requests. These signals range from simple predefined instructions directly forwarded to components, to more elaborate requests requiring limited processing, matching logic, or event triggering. They remain less computationally demanding than advanced cryptographic signals. | 2 |
| System state data require minimal update logic. The state of components is managed through periodic readings and basic logic. Updates are frequent but contain minimal information. | 3 |
| Signals that verify the integrity and accuracy of data. It includes validation logic. The amount of data exchanged is significant, as detailed information is transmitted to ensure integrity. | 6 |
| Signals that require more complex processing, including decoding, conversion, and signal validation. Conversion requires intensive operations, and authentication is crucial for security. These signals include rich data that must be transmitted and verified. | 10 |

Table 3.5: Classification of signals and DMIPS value

## 3.5 ECUs and Database Modeling for System Optimization

### 3.5.1 ECUs: Definition, Functions, and FPGA Perspectives

Electronic Control Units constitute the computational core of modern automobiles, enabling the control of safety, powertrain, comfort, and infotainment functions. Each ECU is essentially a small embedded computer that integrates a microcontroller or processor,

memory, input/output interfaces, communication transceivers, and diagnostic mechanisms,an example is shown in Figure 3.21. It processes sensor data, executes control algorithms in real time, and sends commands to actuators to ensure that the vehicle behaves according to its design specifications. The growing number of ECUs in vehicles,often several dozen in premium cars,reflects both the increasing complexity of automotive functions and the need for modularity and specialization in system design. [29]
From a hardware perspective, an ECU typically includes volatile and non-volatile memory, mixed-signal interfaces, and dedicated safety elements such as watchdogs or error-correcting code. On the software side, it usually runs a real-time operating system, communication stacks, and application software tailored to the specific domain. This dual hardware–software integration enables the ECU to deliver deterministic performance while also ensuring compliance with safety standards such as ISO 26262 [30].
Because no single ECU can centralize all functionality, in-vehicle communication networks are essential for coordinating data exchange among them. Several standardized technologies coexist: LIN provides low-cost communication for simple body functions at modest data rates; CAN and CAN FD serve as robust, fault-tolerant buses for real-time powertrain and chassis control; and Automotive Ethernet has emerged as the high-bandwidth backbone for data intensive applications such as cameras, radar, and infotainment. Each of these networks offers distinct trade-offs in terms of determinism, bandwidth, cost, and scalability, and they are typically combined in layered architectures to balance vehicle requirements.

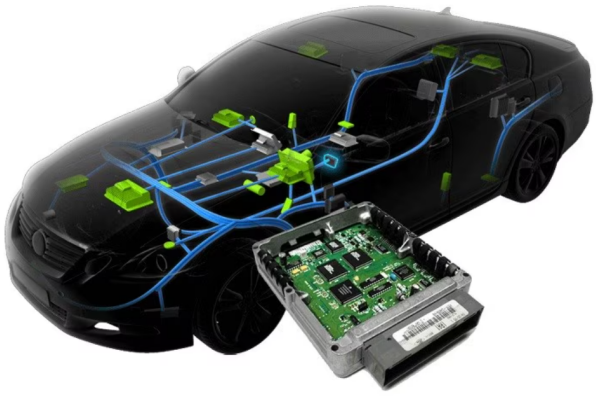

Figure 3.21: Example of car's ECU [11]

At the hardware level, an ECU can be understood as a compact embedded computer, specifically engineered to operate reliably under the demanding conditions of the automotive environment. At its core lies a microcontroller unit or processor, often based on architectures such as ARM, Infineon TriCore, or PowerPC. This processing element is responsible for executing real-time control algorithms, managing communication protocols, and coordinating diagnostic routines. In more advanced applications, particularly those requiring high levels of parallel computation, the microcontroller may be replaced or complemented by multicore processors or FPGA-based System-on-Chip devices, which

offer superior flexibility and hardware-level acceleration.
At its core, the processor is supported by a memory architecture composed of Flash, EEPROM, and SRAM, ensuring deterministic access for real-time control. Interfaces such as analog-to-digital and digital-to-analog converters, together with PWM drivers, connect the ECU to sensors and actuators, translating environmental data into precise control actions. Field-Programmable Gate Arrays (FPGAs) represent a class of integrated circuits whose internal hardware configuration can be reprogrammed after manufacturing. Unlike fixed-function microcontrollers, FPGAs provide a reconfigurable architecture that allows engineers to tailor computational resources to specific applications. This flexibility, combined with their capacity for massively parallel execution, makes them particularly suitable for complex, real-time automotive function. [31]
In the automotive domain, FPGA-based ECUs are increasingly adopted for tasks that require high-performance computation, adaptability, and hardware-level safety mechanisms, applications include sensor fusion in advanced driver-assistance systems (ADAS), implementation of safety-critical communication protocols, electric powertrain control, and high-speed in-vehicle networking. The ability to update or reconfigure FPGA logic even after deployment ensures longer product lifecycles and facilitates compliance with evolving industry standards.In modern vehicle architectures, FPGA devices are also used to support complex operations such as AI inference and sensor fusion. Their parallel computing nature and flexibility make them well suited for interfacing with a wide range of sensors, cameras, and displays in automotive systems [31].A further benefit is their deterministic timing characteristics, essential for safety-critical systems controlled by standards such as ISO 26262. FPGAs can provide lower latencies and greater throughput than software-based ECUs and still be coupled with embedded processors using hybrid System-on-Chip (SoC) designs. In addition, Intel and AMD have recently highlighted the role of FPGAs in automotive applications, with device families qualified for automotive-grade operation.

### 3.5.2 Design and Structure of the ECU Database

Based on the advantages previously discussed, the decision was made to construct a dedicated database of FPGA- and SoC-based ECUs, focusing on devices explicitly recommended for automotive use. The database presented in Table 3.6 includes solutions provided by two of the main semiconductor manufacturers in this domain, Intel and AMD, both of which offer product lines qualified for the stringent requirements of the automotive environment. [32,33] The technical specifications of each ECU were collected directly from the official product handbooks and datasheets, ensuring reliability and alignment with vendor documentation. Several parameters were selected as the basis for evaluation. First, computational performance was quantified using DMIPS values, which serve as a standard benchmark for measuring processing capability. [34, 35] Second, the communication interfaces available on each ECU were considered, with particular attention to the presence or implementability of CAN and LIN bus connectivity, For simplicity, each additional protocol interface has been assumed to incur a cost of 15$.
To evaluate the computational performance of the selected ECUs, their programmable

nature was leveraged. Unlike traditional microcontrollers, FPGA platforms allow the integration of multiple softcore processors, which can be instantiated in parallel to increase processing capacity. In principle, the number and configuration of these softcores can vary depending on the design choices and the specific application requirements. However, for the purpose of this thesis and in order to provide a consistent basis of comparison across different devices, the analysis considered the maximum configuration, i.e., the scenario in which each ECU is programmed with the largest number of softcores supported by its architecture. This assumption ensures that the database reflects the upper bound of computational load achievable by each device, simplifying the benchmarking process while still capturing the scalability advantages of FPGA-based solutions. Finally, cost and mass data for each ECU was collected from commercial distributors such as Digipart and octopart..., providing a market oriented perspective that complements the technical evaluation.

| **ECU Model** | **Part Number** | **Protocol Interface** | **Add. Prot. Int.** | **Comp. load [DMIPS]** | **Cost [$]** | **Max Cost [$]** | **Mass [Kg]** |
|---|---|---|---|---|---|---|---|
| Artix 7 XA FPGA | Arty A7-35T | 1 ET<br>1 LIN | 4 | 211.14 | 140 | 200 | 0.883 |
| MAX 10 | DK-DEV-10M50-A | 2 ET | 10 | 85.722 | 196 | 346 | 1.035 |
| Zynq 7000 XA SoCs | AX7Z010B | 2 ET<br>2 CAN<br>2 LIN | 9 | 419.52 | 198 | 333 | 1.000 |
| Spartan 7 FPGA | AX7050 | 1 ET<br>1 LIN | 4 | 369.84 | 198 | 258 | 1.300 |
| Zynq 7000 XA SoCs | AX7Z020B | 2 ET<br>2 CAN<br>2 LIN | 9 | 1048.8 | 238 | 373 | 1.200 |
| Artix 7 XA FPGA | Arty A7-100T | 1 ET<br>1 LIN | 4 | 1055.7 | 262 | 322 | 0.913 |
| Zynq UltraScale+ SoC | AXU3EGB | 2 ET<br>2 CAN<br>2 LIN | 0 | 830 | 449 | 449 | 1.300 |
| Cyclone V E | DK-DEV-5CEA7N | 1 ET | 10 | 440.856 | 817 | 967 | 1.447 |
| Spartan 7 FPGA | EK-S7-SP701-G (SP701) | 2 ET<br>1 LIN | 6 | 739.68 | 836 | 926 | 2.347 |
| Zynq UltraScale+ SoC | AXU4EVB-P | 2 ET<br>2 CAN<br>2 LIN | 0 | 830 | 885 | 885 | 1.200 |
| Cyclone IV GX | DK-DEV-4CGX150N | 1 ET | 21 | 305.208 | 963 | 1278 | 1.700 |

| **ECU Model** | **Part Number** | **Protocol Interfaces** | **Add. Prot. Int.** | **Comp. load [DMIPS]** | **Cost [$]** | **Max Cost [$]** | **Mass [Kg]** |
|---|---|---|---|---|---|---|---|
| Cyclone V GT | DK-DEV-5CGTD9N | 1 ET | 10 | 881.712 | 966 | 1116 | 2.374 |
| Versal AI Edge XA | VD100 | 2 ET<br>2 CAN<br>1 LIN | 0 | 885 | 1391 | 1391 | 1.200 |
| Cyclone V SX SoC | DK-DEV-5CSXC6N/ES | 3 ET<br>1 LIN<br>1 CAN | 10 | 2230 | 1595 | 1745 | 2.778 |

Table 3.6: ECU discrete database

Building on the developed ECU database, a cost-performance model was constructed using the Curve Fitter tool in MATLAB, the objective was to derive a continuous relationship between computational performance, communication capabilities, and cost. In this framework, the x and y axes represent, respectively, the DMIPS values (as a proxy for computational capacity) and the total number of supported interfaces, while the z-axis corresponds to the estimated cost of the ECU.
To capture the dependency on communication capabilities, each ECU entry in the database was replicated according to the number of possible interfaces. For instance, if a given ECU natively supports two fixed interfaces and allows the implementation of up to three additional ones, four virtual configurations were generated: one with two interfaces, one with three, one with four, and one with five. In order to account for the cost increase associated with additional interfaces, a linear surcharge of 15 $ per interface was applied, while the baseline price was taken from vendor or distributor data, each incremental interface was modeled as an additive cost, ensuring that the final fitted curve incorporates both computational performance and connectivity as cost drivers. The cost curve was generated by considering all possible configurations of each ECU with respect to its available interfaces. As illustrated in Figure 3.22, each dot in the plot corresponds to a single ECU configuration, highlighting the relationship between computational performance, interface count, and cost. The resulting curve presented in Figure 3.22 provides a synthetic but representative model of the trade-offs between performance, communication flexibility, and economic investment, which serves as a valuable input for subsequent optimization of ECU allocation. The polynomial regression carried out in `MATLAB` produced the following analytical cost model:

$$\text{Cost}(x, y) = p_{00} + p_{10}x + p_{01}y + p_{20}x^2 + p_{11}xy \tag{3.1}$$

Where the coefficients were estimated as:

$$
\begin{aligned}
p_{00} &= 71.51 \\
p_{10} &= 1.184 \\
p_{01} &= 15 \\
p_{20} &= -0.0002302 \\
p_{11} &\approx 0 \quad \text{(fixed at bound)}
\end{aligned}
$$

The accuracy of the model is quantified through the **coefficient of determination** ($R^2$)= 0.7807 and the **root mean squared error** (RMSE) = 244.

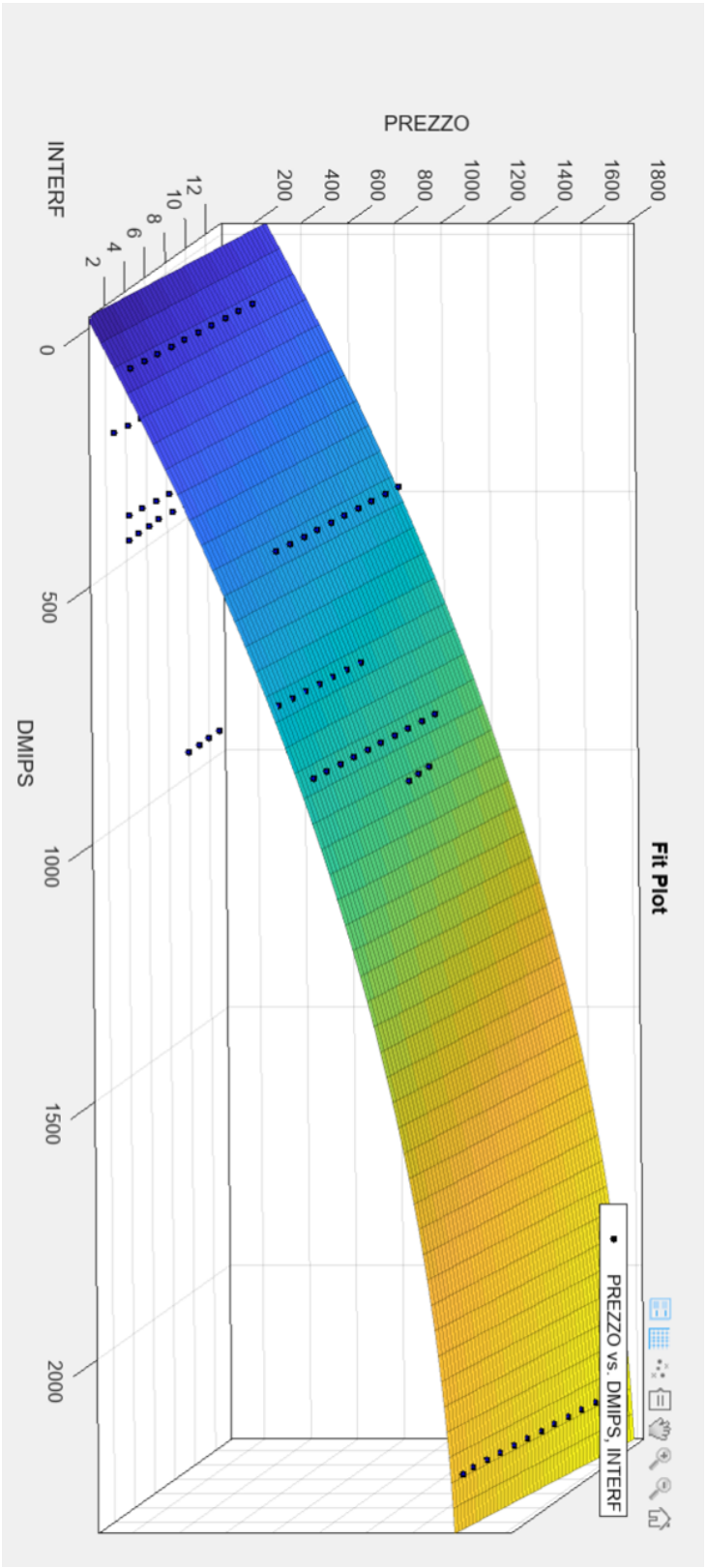


Figure 3.22: Cost-performance-interface plot

## 3.6 Optimization of the Vehicle Electronic Architecture

The optimization process developed in this project stems directly from the system model created in Cameo Systems Modeler, then exported as three well-formed CSV tables. Each table has its purpose, in that the first table outlines the exchanges between varying system functions, with the second table indicating each function computational weight as DMIPS; while the third table establishing the weight of the communication signals. With this information the optimizer can determine how functions should be allocated on the ECUs, the associated computational burden, and find the best configurations for the vehicle's network.
While discussing the problem formulation, the types of design variables are directly related to the number of functions and the number of groups, presented in Table 3.2. The functions to be allocated will also be subject to constraints that make sure that the overall architectures are feasible. For example, there will only be a certain amount of functions that each ECU can host, tightly coupled group of functions presented in Chapter 3.4.3 can be consolidated together and not distributed on multiple ECU, this constraint is implemented in Python, as illustrated in Figure 3.23 where *G_F1* stands for the first function and the number in blue is related to the assigned group. Even limits relative to functionality originate from ECU database, like limits on maximum computation load, and the number of interfaces that can be supported.
The Python code is flexible: there are flags at the beginning of the solution to allow the user to select their optimization modes, shown in Figure 3.24, for example, the user can decide to work with a continuous database, with performance–cost curves, which are interpolated, or a discrete database, with catalog entries. The user can also decide to do a single-objective optimization (cost is minimized) or multi-objective optimization (trade-off between cost and weight).

```
functions_allocation = {
    "G_F1": 1,
    "G_F2": 1,
    "G_F3": 2,
    "G_F4": 2,
    "G_F5": 2,
    "G_F6": 2,
    "G_F7": 2,
    "G_F8": 2,
    "G_F9": 2,
    "G_F10": 2,
    "G_F11": 9,
    "G_F12": 3,
    "G_F13": 4,
    "G_F14": 4,
    "G_F15": 5,
    "G_F16": 1,
    "G_F17": 1,
    "G_F18": 5,
    "G_F19": 5,
    "G_F20": 5,
    "G_F21": 5,
    "G_F22": 10,
```

Figure 3.23: Function allocation

```
# Set the following flag to choose among two possible tool modes:
#     - True: Perform the optimization
#     - False: Calculate solution only for the initial configuration
FLAG_Optimization = True


# Set the following flag to choose among two possible tool modes:
#     - True: Compute cost using the regression curve
#     - False: Compute cost using the discrete database
FLAG_CONTINUOUS_DATABASE = False

# Set the following flag to choose among two possible tool modes:
#     - True: Compute only cost
#     - False: Compute cost and mass
FLAG_Singleobjective = False
```

Figure 3.24: Optimization modes selection

The Multidisciplinary Feasible (MDF) form of architecture was selected as the reference architecture based upon the structure of the optimization problem because it guarantees that multidisciplinary analysis is consistent at each optimization iteration, and is consistent in the sense that coupling variables are solved completely and consistently prior to the acquisition of information by the optimizer. This is especially desirable because it guarantees that every candidate solution that is evaluated in front of the optimizer is truly a feasible system, which is especially valuable from an engineering perspective, where it is critical that the entire system is consistent [10]. In addition, MDF formulations typically have smaller optimization problems than the other architectures, such as SAND or IDF, which will reduce computing cost.
The optimization process was segmented into several disciplines each responsible for an area of system evaluation in this framework;in order to clearly show the flow of information, the workflow was depicted in the XDSM diagram in Figure 3.25 showing the input and output for each disciplines and the sequence of action for the optimization; for the multi objective optimization the mass discipline has been added after the cost evaluation.

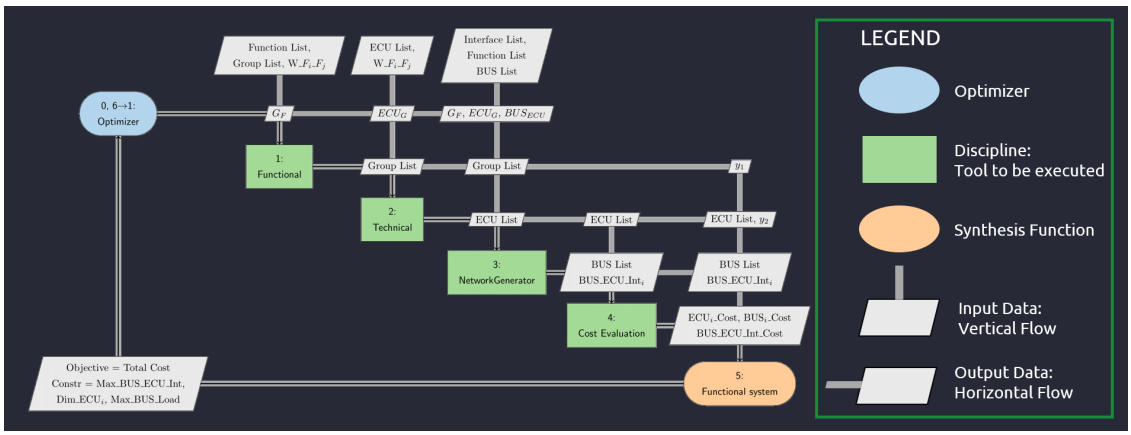


Figure 3.25: Case study's XDSM diagram

### 3.6.1 Functional discipline

The functional discipline is the kickoff point of the optimization workflow because it converts the SysML model exported from Cameo into a compact, computable representation of what the system does, and how the functions connect. The functional discipline takes those inputs and constructs a function to function interaction matrix (the matrix shown in Figure 3.26). Each function is represented by a unique cell on the main diagonal, F1 for example, each off-diagonal entry captures pairwise exchanges: an "X" indicates that there is one signal between the function in row i and the function in column j.
Apart from producing the interaction matrix, this discipline is also responsible for the assignment of the various functions to their groups. As described above, these groups were predetermined and imposed before optimization, meaning that certain sets of functions remain co-located and are treated as inseparable units in subsequent optimization steps.

The function groups shown in Figure 3.26 , are represented as grey squares;the function groups are important because they enforce the constraint that all functions must belong to a group and thus all functions in a group must be put onto the same ECU.

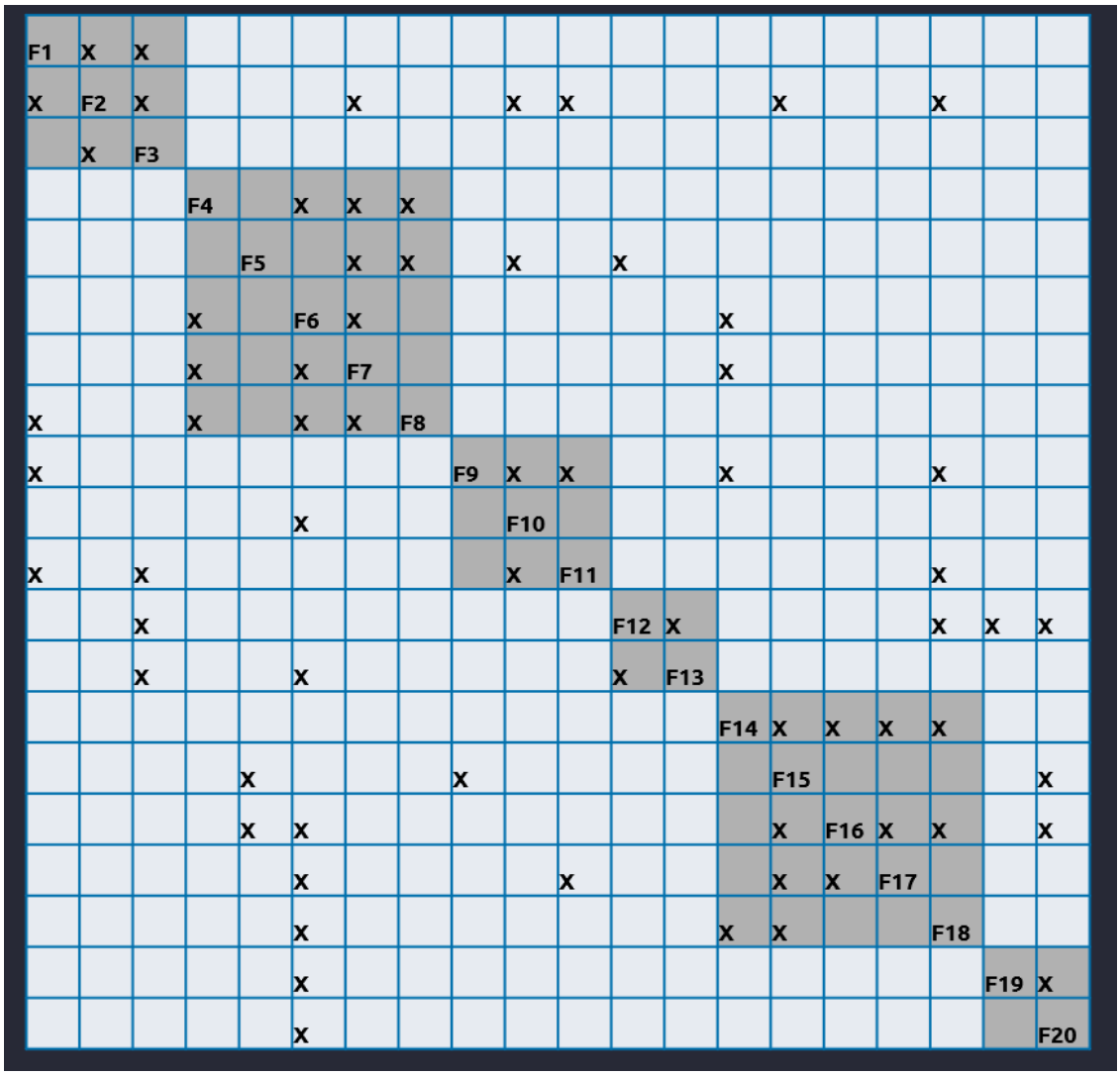


Figure 3.26: Example of functional matrix model

### 3.6.2 Technical discipline

The Technical Discipline is the second discipline of the optimizer; it takes the functional model as input from the last step, namely the functions and function groups, and builds the ECUs list. When mapping, the discipline relates function groups to ECUs while meeting ECU dimension constraints. ECUs are depicted in Figure 3.27 by the colored line border that encompasses the groups, showing that the functions of a group are together.

So as a result of this discipline we will have a first approximation of the functional groups mapped onto ECUs; this sufficient step is the basis for the next, phase of generating the network.

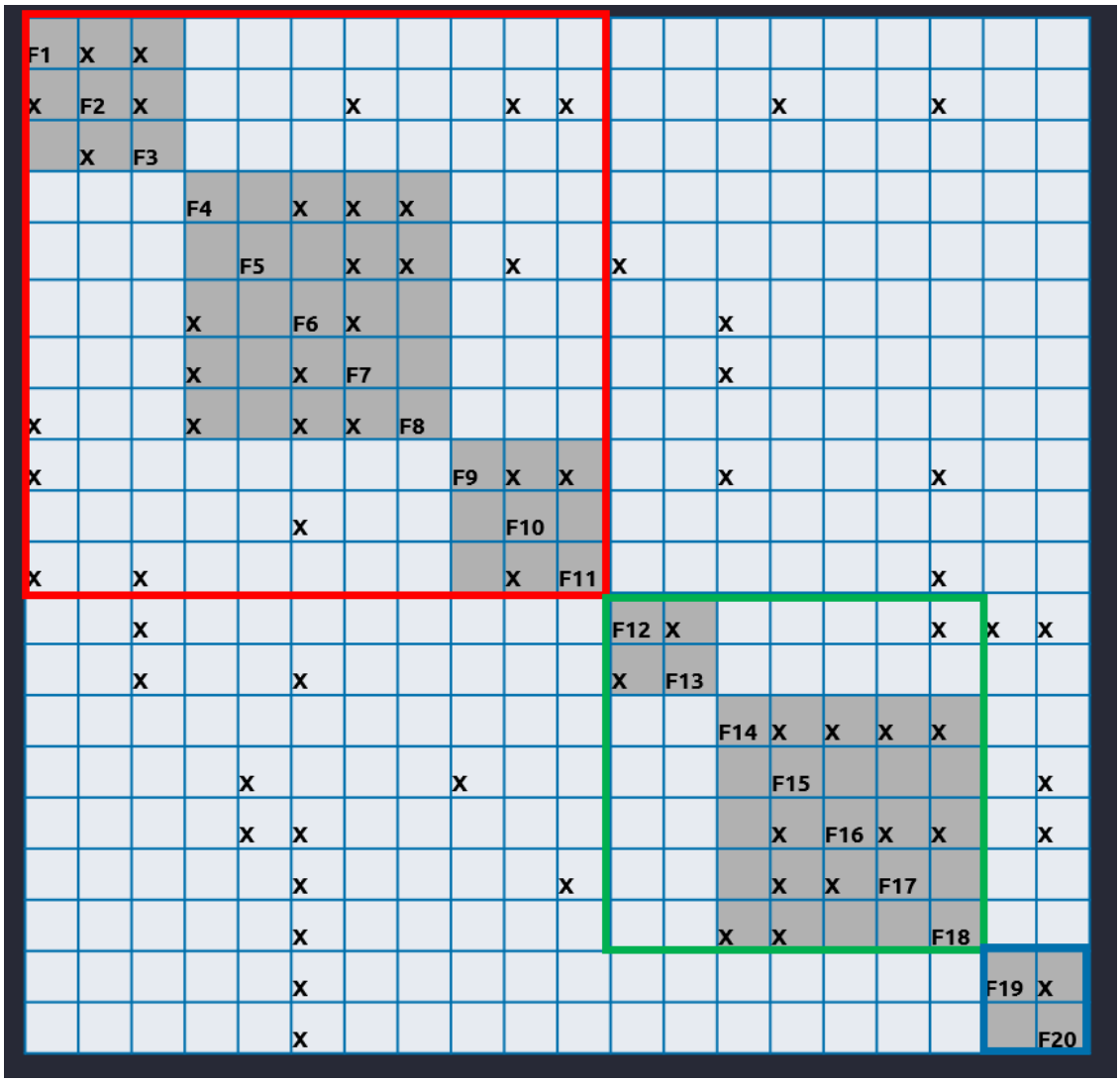


Figure 3.27: Example of technical matrix model

### 3.6.3 Network generator discipline

The third discipline in the optimization workflow is the Network Generation Discipline. At this stage, a listing of communication buses is created, the inputs is the technical model that was created in the previous step, containing all of the functions, ECUs, and relationships. The discipline works to identify where and how to place and configure the communication buses in a way that satisfies the necessary interactions among the ECUs.

It confirms that the functional exchanges identified in the previous disciplines are satisfied by the network infrastructure. The outputs are a complete network configuration,shown in Figure 3.28, of the ECUs and the buses that connect them; looking at the figure, it becomes clear that each communication bus is represented by a distinct color. The X marks in the off-diagonal terms of the matrix indicate the signals exchanged between the ECUs, making explicit the communication links that must be supported by the network.

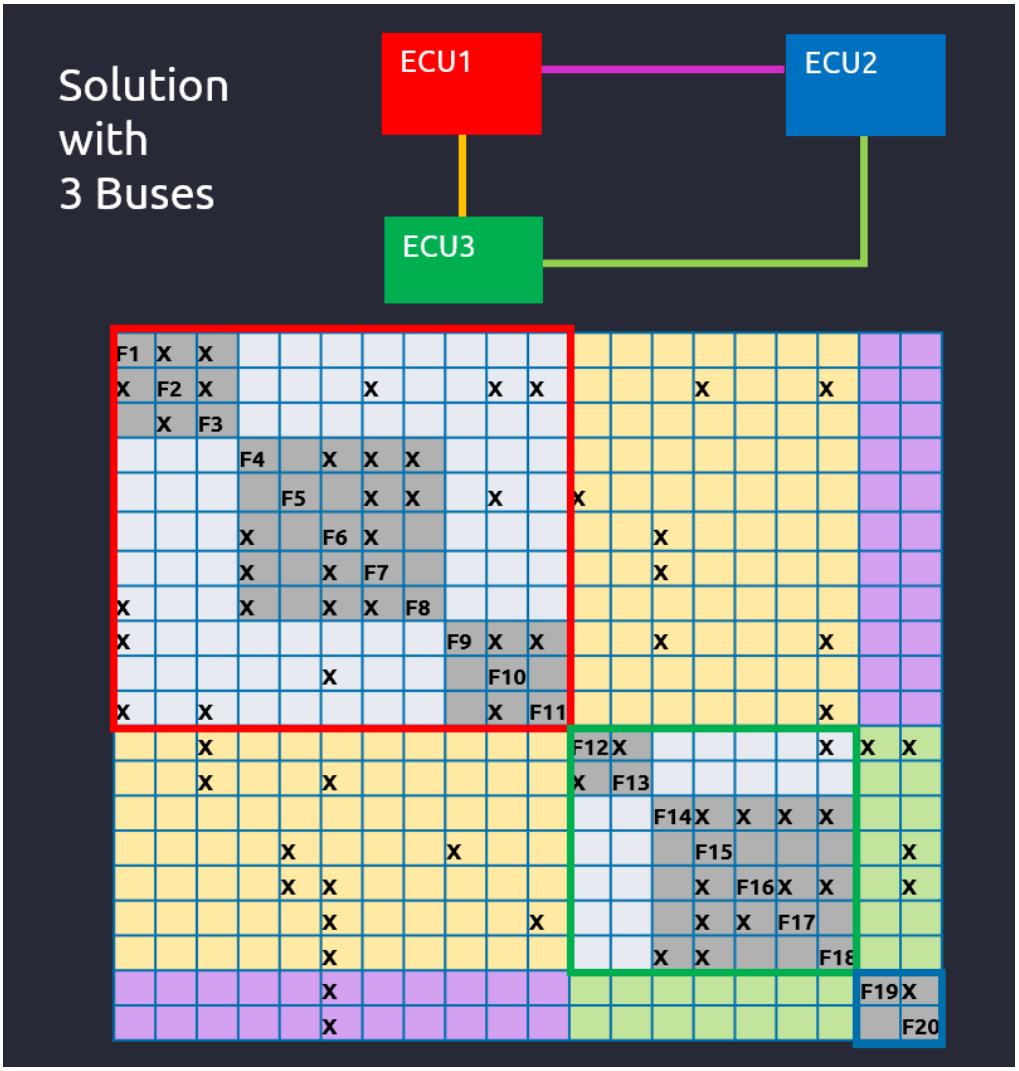


Figure 3.28: Example of network generator matrix model

### 3.6.4 Cost and Mass evaluation discipline

The Cost Discipline plays a central role in the optimization framework, as it evaluates the economic implications of each proposed network configuration. As we noted before, this

discipline can be applied to both single objective optimization (in continuous and discrete formulations) and multi-objective optimization as well. The inputs to Cost Discipline are the network architectures from the previous stage when the full set of ECUs and their corresponding communications buses exist.
The first step, which is the same in all cases, characterizing the basic parameters for each ECU; which are the number and type of communications interfaces, and the computational load in order to satisfy the functions allotted to each ECU. This is accomplished in Python by the function compute ecu from catalogue and returns the number of required interfaces in the configuration of LIN, CAN, or Ethernet, and the computational load of the ECUs,it is calculated as the sum of the computational weight of the internal signals and functions made available to the ECU. So, it returns four outputs which are the number of LIN,CAN,Ethernet interfaces, and the total value of DMIPS required, which is the basis for the cost measurements.
For the **single-objective continuous case** , we plug the interface parameters and computational burden into the polynomial cost function, equation 3.1, we previously established using curve fitting. The output is the estimated cost for each ECU delivered as a continuous function of performance and connection.
In the **single-objective discrete case** the optimization code manually added a catalog of ECU options as shown in Python code 3.1. The algorithm checks to see which ECU model meet all the imposed constraints (for interfaces and computational capacity) and extract a list of possible ECU; then, among the options that meet the constraints, a specific function min cost selects the cheapest option to be included in the architecture.
For the **multi-objective optimization** therefore only the discrete formulation was implemented. Following the same process as for the discrete single-objective case, the procedure remains the same, however, the selection criterion changes: all feasible ECUs were normalized for cost and weight and the ECU with the least combined normalized score was selected as shown in the Python code 3.1 in the function min mass cost.

```
# Databases
NAME_DATABASE = [
    'Arty␣A7-35T','DK-DEV-10M50-A','AX7Z010B','AX7050','AX7Z020B','
       Arty␣A7-100T',
    'AXU3EGB','DK-DEV-5CEA7N','SP701','AXU4EVB-P','DK-DEV-4CGX150N'
       ,
    'DK-DEV-5CGTD9N','VD100','DK-DEV-5CSXC6N/ES','ECU_NOT_REAL'
]
LIN_DATABASE = [1,0,2,1,2,1,2,0,1,2,0,0,1,1,float('inf')]
CAN_DATABASE = [0,0,2,0,2,0,2,0,0,2,0,0,2,1,float('inf')]
ET_DATABASE  = [1,2,2,1,2,1,2,1,2,2,1,1,2,3,float('inf')]
DMIPS_DATABASE =
   [211.14,85.722,419.52,369.84,1048.8,1055.7,830,440.856,739.68,830,

                    305.208,881.712,885,2230,float('inf')]
PIN_DATABASE = [4,10,9,4,9,4,0,10,6,0,21,10,0,10,float('inf')]
COST_DATABASE =
   [140,196,198,198,238,262,449,817,836,885,963,966,1391,1595,10000]
```

```
MASS_DATABASE =
    [0.883,1.035,1,1.3,1.2,0.913,1.3,1.447,2.347,1.2,1.7,2.374,1.2,
2.778,float('inf')]
for i in range(len(DMIPS_DATABASE)):
    if DMIPS_DATABASE[i] >= DMIPS and ET_DATABASE[i] >= ET:
        # SoC case
        if PIN_DATABASE[i] == 0:
            if CAN_DATABASE[i] >= CAN and LIN_DATABASE[i] >= LIN:
                indici.append(i)
                plus.append(0)
        else:
            # FPGA case
            if CAN_DATABASE[i] >= CAN:
                if LIN_DATABASE[i] >= LIN:
                    indici.append(i)
                    plus.append(0)
                else:
                    # Not enough LIN interfaces: use pins for the
                       delta
                    LIN_new = LIN - LIN_DATABASE[i]
                    if PIN_DATABASE[i] >= LIN_new:
                        indici.append(i)
                        plus.append(LIN_new)
            else:
                # Not enough CAN interfaces: use pins for the delta
                CAN_new = CAN - CAN_DATABASE[i]
                if PIN_DATABASE[i] >= CAN_new:
                    PIN_DATABASE[i] = PIN_DATABASE[i] - CAN_new
                    if LIN_DATABASE[i] >= LIN:
                        indici.append(i)
                        plus.append(CAN_new)
                    else:
                        # Use pins for both CAN and LIN deficits
                        LIN_new = LIN - LIN_DATABASE[i]
                        if PIN_DATABASE[i] >= LIN_new:
                            indici.append(i)
                            plus.append(LIN_new + CAN_new)
for k in range(len(indici)):
    idx = indici[k]
    COST_DATABASE[idx] += 15 * plus[k]
    NEW_PIN[idx] -= plus[k]
# Normalize cost and mass (excluding the sentinel inf at the end)
min_cost, max_cost = min(COST_DATABASE[:-1]), max(COST_DATABASE
    [:-1])
NORM_COST = [(x - min_cost) / (max_cost - min_cost) for x in
    COST_DATABASE[:-1]]
NORM_COST.append(COST_DATABASE[-1])
min_mass, max_mass = min(MASS_DATABASE[:-1]), max(MASS_DATABASE
    [:-1])
```

```
NORM_MASS = [(x - min_mass) / (max_mass - min_mass) for x in
    MASS_DATABASE[:-1]]
NORM_MASS.append(MASS_DATABASE[-1])
for i in indici:
    ECU_DA_CONFRONTARE_COST.append(COST_DATABASE[i])
    ECU_DA_CONFRONTARE_NAME.append(NAME_DATABASE[i])
    ECU_DA_CONFRONTARE_MASS.append(MASS_DATABASE[i])
    ECU_DA_CONFRONTARE_COST_NORM.append(NORM_COST[i])
    ECU_DA_CONFRONTARE_MASS_NORM.append(NORM_MASS[i])
return (ECU_DA_CONFRONTARE_COST, ECU_DA_CONFRONTARE_MASS,
         ECU_DA_CONFRONTARE_NAME, ECU_DA_CONFRONTARE_COST_NORM,
        ECU_DA_CONFRONTARE_MASS_NORM)
def min_cost(ECU_DA_CONFRONTARE_COST, ECU_DA_CONFRONTARE_NAME):
    # Find minimum value and index
    minimo = min(ECU_DA_CONFRONTARE_COST)
    pos = ECU_DA_CONFRONTARE_COST.index(minimo)
    nome = ECU_DA_CONFRONTARE_NAME[pos]
    return minimo, nome
def min_mass_cost(ECU_DA_CONFRONTARE_COST, ECU_DA_CONFRONTARE_MASS,
                  ECU_DA_CONFRONTARE_NAME,
                     ECU_DA_CONFRONTARE_COST_NORM,
                  ECU_DA_CONFRONTARE_MASS_NORM):
    # Compute normalized sum of cost and mass
    somma_norm = list(map(operator.add,
                          ECU_DA_CONFRONTARE_COST_NORM,
                          ECU_DA_CONFRONTARE_MASS_NORM))
    pos = somma_norm.index(min(somma_norm))
    minimo_costo = ECU_DA_CONFRONTARE_COST[pos]
    nome = ECU_DA_CONFRONTARE_NAME[pos]
    minima_massa = ECU_DA_CONFRONTARE_MASS[pos]
    return minimo_costo, nome, minima_massa
```

Listing 3.1: ECU filtering and cost and mass evaluation

## 3.7 Results

To assess the viability of the optimization framework presented, this study considered three categories of optimization problems: single-objective discrete, single-objective continuous, and multi-objective discrete. For each formulation, several test were run by changing the maximum number of functions that could be allocated to a single ECU. The maximum number of functions to allocate to a single ECU was the parameter examined because of its direct impact on both the scalability of the architecture and the comprehensive understanding of computational workload versus communication needs and cost.All optimization runs took place in the same computational environment so that results would be comparable. The experiments were conducted on a machine equipped with an Intel(R) Core(TM) i5-10310U CPU running at 1.70 GHz (boost up to 2.21 GHz) and 16 GB of RAM.
The optimization problem, shown in equation (3.2)–(3.8), in this work solved using the

Multidisciplinary Feasible (MDF) approach discussed before. Within this formulation, the decision variables include the allocation of groups in the ECUs (ECU_Gj) and the configuration of communication buses (BUS_ECUk_ECUj), while the objectives are the minimization of total cost and, in the multi-objective case, minimization of both cost and weight. Constraints are enforced to guarantee feasibility, including limits on the maximum number of functions per ECU(eq. (3.4)), the computational capacity expressed in DMIPS for ECU (eq. (3.6)) and Buses (eq. (3.5)) and the maximum number of interfaces available (eq. (3.7), eq. (3.8)).

$$\begin{aligned}
\text{minimize} \quad & \text{Cost} \quad \text{or} \quad \text{Cost_Mass} && (3.2)\\
\text{with respect to} \quad & ECU_G_j \quad j = 1, \ldots, N && (3.3)\\
& BUS_ECU_k_ECU_j \quad j, k = 1, \ldots, N \\
\text{subject to} \quad & \text{Max_Dim_ECU} \in \{10{,}12{,}15\} && (3.4)\\
& \text{Bus_load} \leq 100 \text{ Mbit/s} && (3.5)\\
& \text{ECU_Comp_Load} \leq 2230 \text{ DMIPS} && (3.6)\\
& \text{ECU_Interf} \leq 13 && (3.7)\\
& \text{ECU_Et} \leq 3 && (3.8)
\end{aligned}$$

Where:

- ECU_G represent the group of function previously presented
- N represent the number of groups
- BUS_ECU_ECU represent the bus that connect the ECUi with the ECUj
- Max_Dim_ECU represent the maximum number of functions that can be allocated in each ECU
- ECU_Comp_Load represent the maximum computational load for each ECU
- Bus_load represent the value of the maximum computational load for each BUS
- ECU_Interf represent the maximum number of possible protocol interfaces (LIN+CAN) for each ECU
- ECU_Et represent the maximum number of possible Ethernet interfaces for each ECU in the network

For the optimization, the parameters listed in Table 3.7 were selected:

| **Parameter** | **Value** |
|---|---|
| Number of functions | 50 |
| Function weight | from 3 to 30 DMIPS |
| Function interface weight | from 1 to 6 DMIPS |
| Optimization algorithm | `Pymoo_GA` |
| Pop size | 1000 |

Table 3.7: Optimization setup

### 3.7.1 Single-objective optimization discrete database

The first set of experiments focused on discrete single-objective optimization, in which ECU configurations were directly selected from the database. Three different constraints on the maximum number of functions per ECU were tested: 10, 12, and 15. For the case with a maximum of 10 functions per ECU, the optimizer generated a solution consisting of six ECUs and two communication buses, with a total cost of 1029.75$ and an execution time of 1 hour, 03 minutes, and 54 seconds; the network configuration is shown in Figure 3.29. It should be noted that, based on the assumptions defined by the functional groups in Table 3.2, the minimum feasible number of ECUs was six, a condition that was correctly satisfied by the optimizer. The allocation process ensured that all functions were mapped to ECUs while always respecting the grouping constraints, thus preserving the integrity of the predefined functional clusters. Moreover, the optimizer try to minimize the number of buses, since reducing interconnections directly contributes to lowering overall cost. This behavior is evident in the resulting configuration, where the number of buses remains low but still meets all communication requirements. In particular, one of the buses connects ECUs 10 and 11 simultaneously with ECU 6, providing a cost-effective solution that balances functional allocation with minimal networking overhead, and the other two buses connecting ECU 3 with ECU 10 and ECU 6.

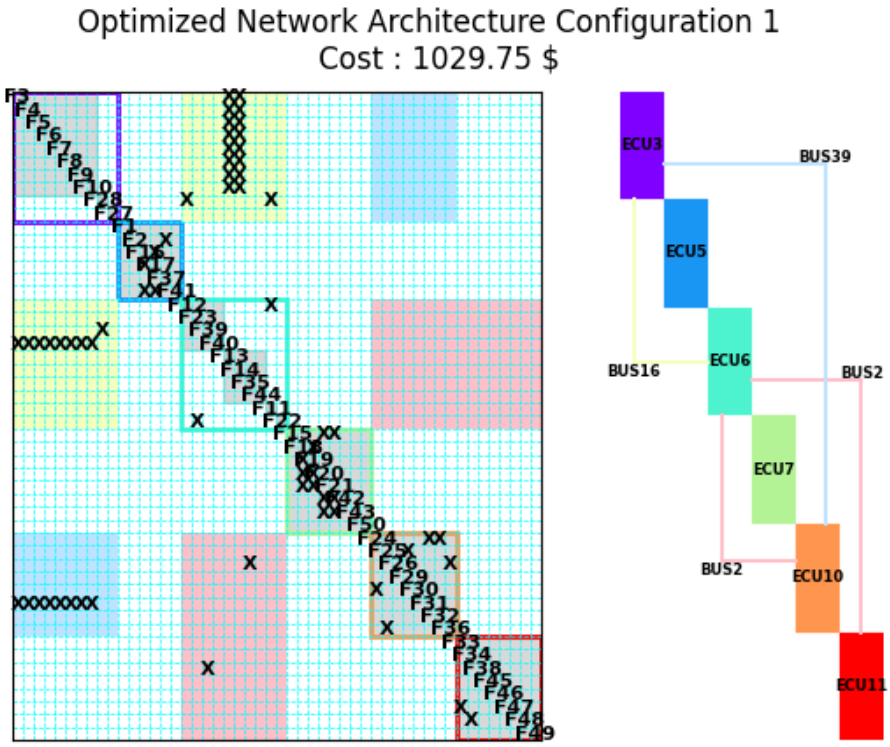


Figure 3.29: Single-objective optimization discrete database max dim 10

| ECU | Cost ($) | Model |
|---|---|---|
| ECU_3 | 198.00 | AX7Z010B |
| ECU_5 | 140.00 | Arty A7-35T |
| ECU_6 | 198.00 | AX7Z010B |
| ECU_7 | 140.00 | Arty A7-35T |
| ECU_10 | 198.00 | AX7Z010B |
| ECU_11 | 155.00 | Arty A7-35T |
| BUS_2 | 0.25 | CAN |
| BUS_16 | 0.25 | CAN |
| BUS_39 | 0.25 | CAN |
| **Total** | **1029.75** | – |

Table 3.8: ECU cost and model summary single-objective optimization discrete database max dim 10

When the limit was increased to 12 functions per ECU, the optimizer identified a more compact solution with five ECUs interconnected by five buses, achieving a lower overall cost of 889.75 $ and requiring 23 minutes and 56 seconds to compute. The network

configuration is shown in Figure 3.30 and the cost and model of the ECU are illustrated in Table 3.9.

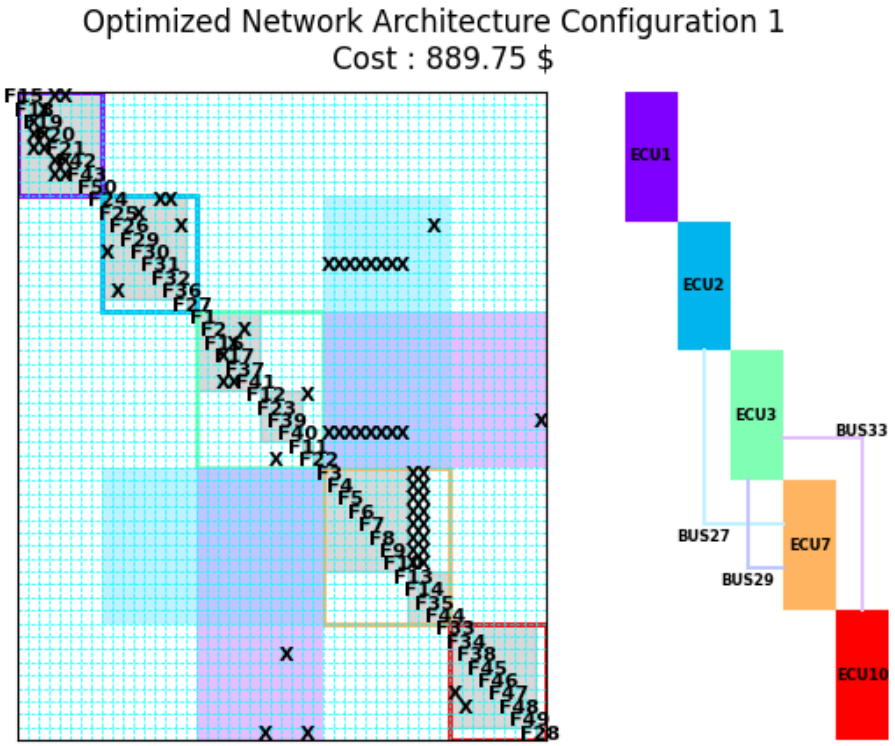


Figure 3.30: Single-objective optimization discrete database max dim 12

| ECU | Cost ($) | Model |
|---|---|---|
| ECU_1 | 140.00 | Arty A7-35T |
| ECU_2 | 198.00 | AX7Z010B |
| ECU_3 | 198.00 | AX7Z010B |
| ECU_7 | 198.00 | AX7Z010B |
| ECU_10 | 155.00 | Arty A7-35T |
| BUS_27 | 0.25 | CAN |
| BUS_29 | 0.25 | CAN |
| BUS_33 | 0.25 | CAN |
| **Total** | **889.75** | – |

Table 3.9: ECU cost and model summary single-objective optimization discrete database max dim 12

Finally, for the case with a maximum of 15 functions per ECU, the optimizer produced an even more compact architecture of four ECUs and one bus, one of which connected

three ECUs simultaneously. This configuration achieved the lowest cost of the three scenarios, 774.25 $, with an execution time of 42 minutes, and 14 seconds.This network configuration is shown in Figure 3.31 and the cost and model of the ECU are illustrated in Table 3.10. The data show a clear trend: with a larger maximum ECU dimension, the number of ECUs will decrease, which fundamentally reduces the total cost of the network. Large ECUs can hold more functions and subsequently fewer devices, which provides a simpler network topology.

| ECU | Cost ($) | Model |
|---|---|---|
| ECU_2 | 238.00 | AX7Z020B |
| ECU_4 | 198.00 | AX7Z010B |
| ECU_8 | 198.00 | AX7Z010B |
| ECU_9 | 140.00 | Arty A7-35T |
| BUS_14 | 0.25 | CAN |
| **Total** | **774.25** | – |

Table 3.10: ECU cost and model summary single-objective optimization discrete database max dim 15

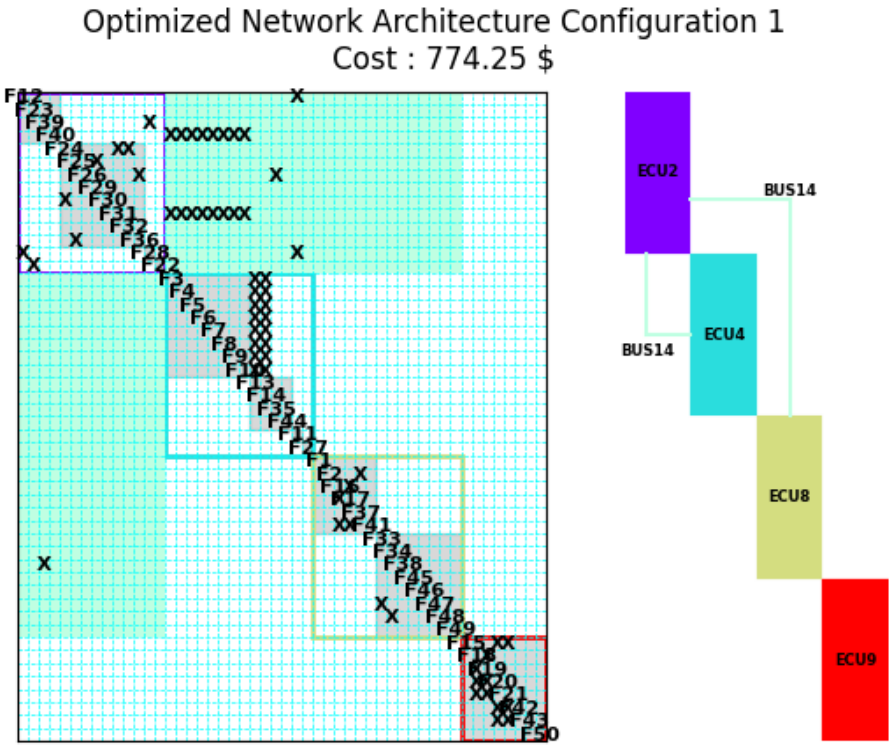


Figure 3.31: Single-objective optimization discrete database max dim 15

### 3.7.2 Single-objective optimization continuous database

The second set of experiments focused on the single-objective continuous optimization; in this process, the price of each ECU was predicted using the polynomial cost function described earlier. The same three constraints on the maximum number of functions per ECU as in the discrete case were employed in this experiment: 10, 12, and 15.
Figure 3.32 depicts the configuration with a maximum of 10 functions per ECU. The optimizer returned a solution of 6 ECUs and 4 buses, and the cost amounted to 2324.00 $ while the execution time was 1 hour, 24 minutes, and 06 seconds. The buses used in this configuration spread out the functional interactions of the ECUs, and all functional interactions were satisfied. The overall effect was to make a more interactive network compared to the discrete solution.
When the maximum number of functions was raised to 12 (Figure 3.33), the optimizer again provided a solution, which was a more constrained solution of 5 ECUs and 2 buses. Here, the total cost was 2004.55$ with a shorter execution time of 51 minutes and 37 seconds.
Finally, in the case with a maximum of 15 functions per ECU (Figure 3.34), the optimizer identified a configuration of four ECUs with three buses, achieving the lowest cost among the three continuous scenarios: 1,864.69 $. This run required 55 minutes, and 05 seconds, due to the broader search space explored by the optimizer when larger ECUs are permitted.

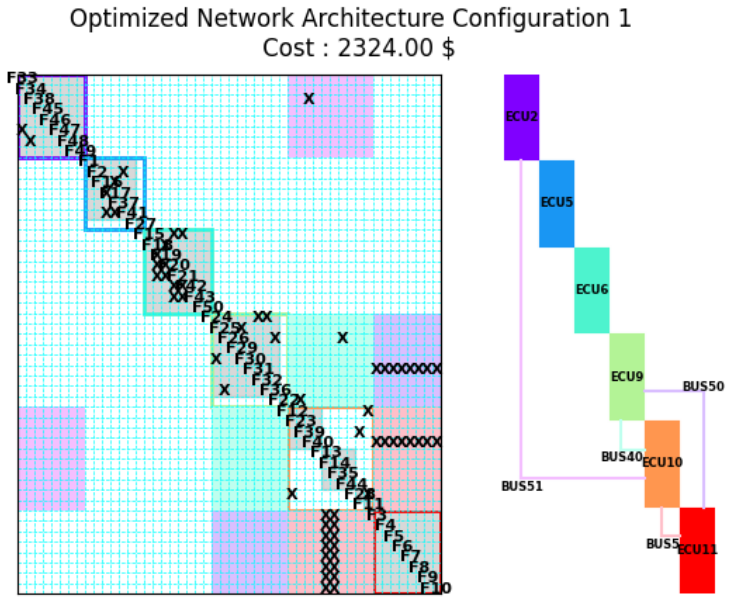


Figure 3.32: Single-objective optimization continuous database max dim 10

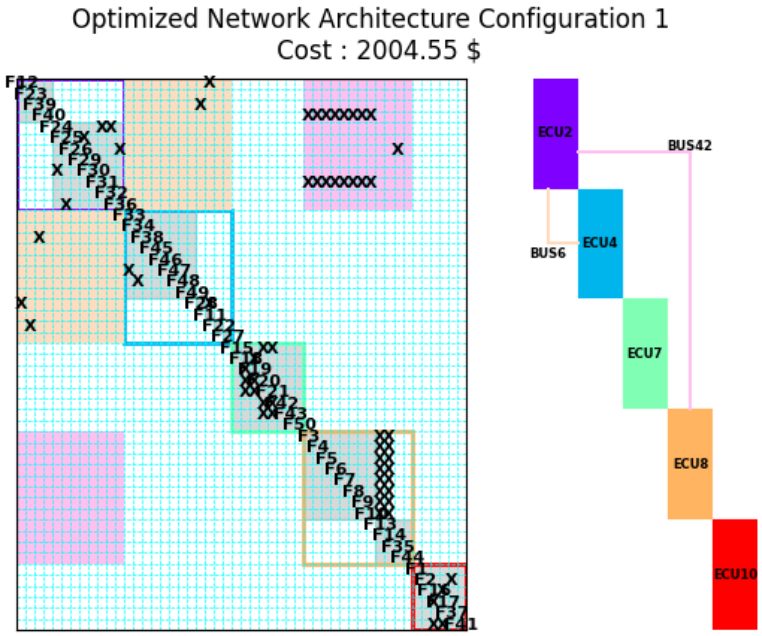


Figure 3.33: Single-objective optimization continuous database max dim 12

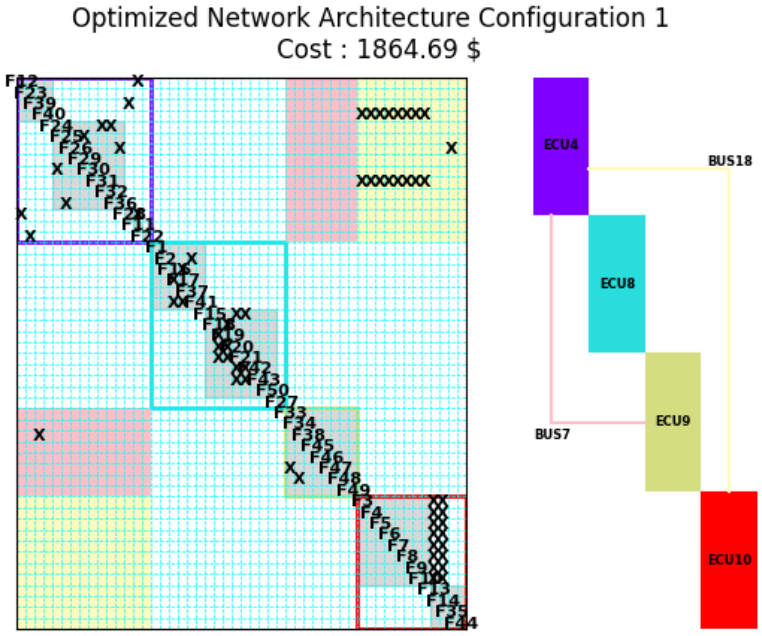


Figure 3.34: Single-objective optimization continuous database max dim 15

An important insight of this result is that the optimization cost values from the continuous optimization are always greater than those from the discrete optimization; this difference is a result of how the ECU cost is modeled with curve fitting in the continuous optimization: the polynomial equation does not perfectly model the cost trends of the actual ECU costs of the dataset, and the optimization yields a cost which has deviations that yield a higher cost. As such, while the continuous model remains useful in understanding and demonstrating the applied relationship between cost and performance, it primarily enables the optimization process to capture a smoother and more continuous representation of system behavior. However, this approach does not provide the same predictive accuracy of absolute cost values as the catalog-based discrete formulation. Despite this limitation, the experiments demonstrated the feasibility of applying the continuous methodology, confirming that it can still generate valid and consistent architectures. Compared to the discrete case, the continuous optimization tends to produce solutions with a higher number of communication buses. This behavior suggests that while the continuous model can capture general cost-performance trends, its less constrained nature may lead to network configurations that rely on additional interconnections, thereby increasing overall cost.

### 3.7.3 Multi-objective optimization discrete database

In the case with a maximum of 12 functions for each ECU, the optimizer was run using a multi-objective setup whereby both cost and weight were minimized at the same time. In contrast to the single-objective cases, the results include a Pareto front that captures the best trade-offs between the two objectives.
In the chart 3.35 produced by the optimizer, we can see the full picture of the configurations that have been analyzed; there are three categories of points used in this chart to describe the possible configurations: Pareto-optimal points, Pareto-dominated points, and non-feasible points. The points that are defined as Pareto-optimal are the points that are designated as optimal solution in the results table (Table 3.11), they constitute the Pareto fronts, and provide the best possible compromise between cost and weight. In contrast to the Pareto-optimal points, the Pareto-dominated points specify that there are valid ECU network configuration points that are strictly worse than Pareto-optimal points as they incur additional costs or increase weight without providing additional value. The non-feasible points represent types of configurations that violate at least one constraint established at the outset of the entire process.
The second chart 3.36 shows a zoom of the area of interest indicated in the first chart 3.35, and this view provides a more comprehensive inspection of the trade-offs in the most salient region of cost and weight. The zoomed-in view also depicts the utopia point, which represents the theoretical configuration that minimizes both objectives simultaneously. It is not always possible to achieve the Utopia point in practice; however, it serves as a useful benchmark to assess how close the obtained solutions are to the ideal. In our case, the Utopia point has objective values of 904.95 $ and 4.662 Kg. Among the configurations generated by the optimizer on the Pareto front (indicated by the blue points), the closest configuration to the Utopia point is the Configuration 2 with values of 953.5 $ and 4.769 Kg,shown in Table 3.11.

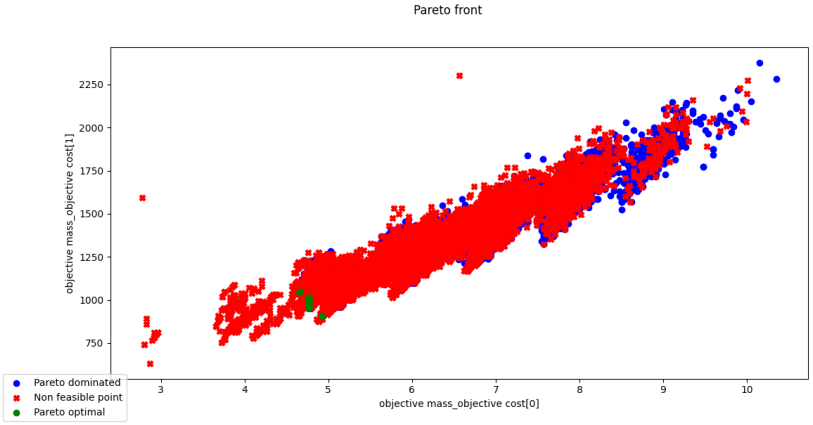


Figure 3.35: Pareto front of the multi-objective optimization continuous database max dim 12

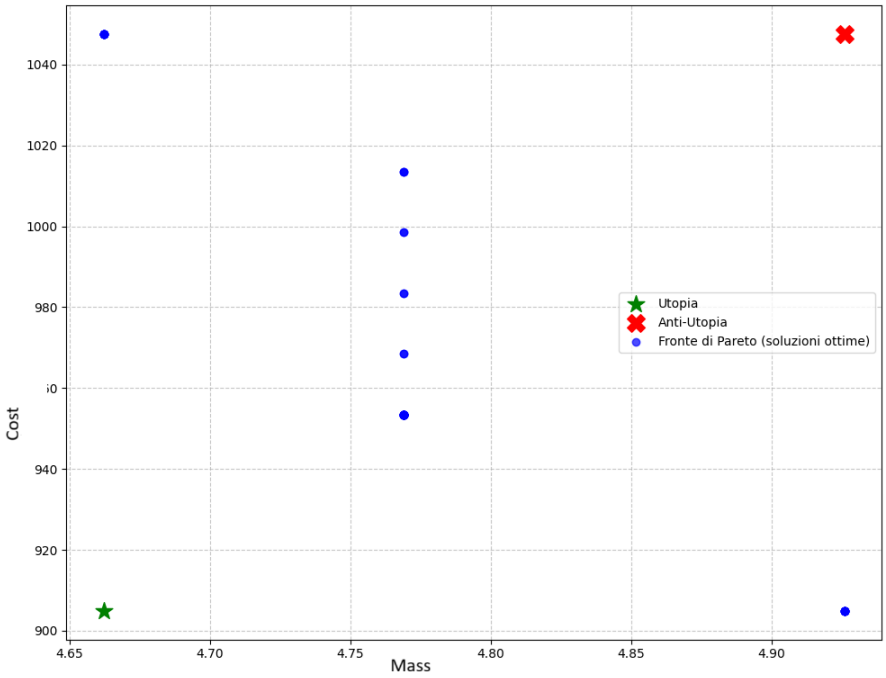


Figure 3.36: Zoom of pareto-front of the multi-objective optimization continuous database max dim 12

In order to better illustrate the trade-offs captured by the Pareto front, two representative configurations were selected from Table 3.11 and are presented in Figures 3.37 and 3.38. Configuration 2, shown in Figure 3.37, represents the solution closest to the Utopia point,achieving the most balanced trade-off between mass and cost,in contrast, Figure 3.38 illustrates the configuration 7 with the lowest overall cost.

| Configuration | Cost [$] | Mass [Kg] |
|---|---|---|
| 1 | 1047.45 | 4.662 |
| 2 | 953.5 | 4.769 |
| 3 | 968.5 | 4.769 |
| 4 | 983.5 | 4.769 |
| 5 | 998.5 | 4.769 |
| 6 | 1013.5 | 4.769 |
| 7 | 904.95 | 4.926 |

Table 3.11: Pareto-optimal configuration (max dim 12)

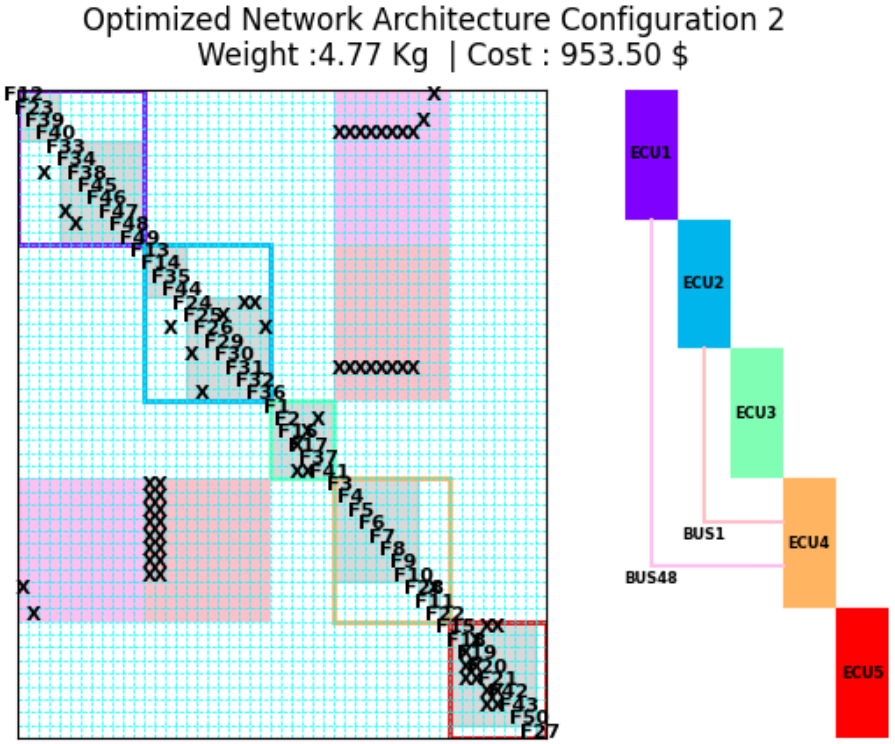


Figure 3.37: Multi-objective Optimized Network Architecture Configuration 2 (max dim 12)

| ECU | Cost ($) | Model | Mass (kg) |
|---|---|---|---|
| ECU_1 | 198.00 | AX7Z010B | 1.000 |
| ECU_2 | 277.00 | Arty A7-100T | 0.913 |
| ECU_3 | 140.00 | Arty A7-35T | 0.883 |
| ECU_4 | 198.00 | AX7Z010B | 1.000 |
| ECU_5 | 140.00 | Arty A7-35T | 0.883 |
| BUS_1 | 0.25 | CAN | 0.045 |
| BUS_48 | 0.25 | CAN | 0.045 |
| **Total** | **953.5** | – | **4.769** |

Table 3.12: ECU costs, models, and masses (Multi-objective max dim 12 Configuration 2)

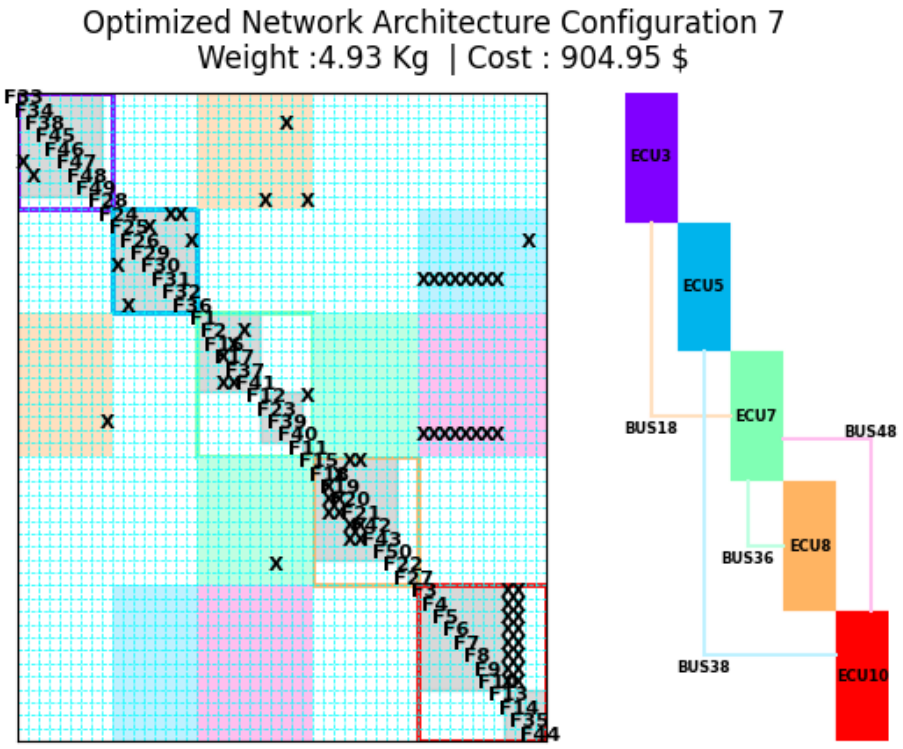


Figure 3.38: Multi-objective Optimized Network Architecture Configuration 7 (max dim 12)

| ECU | Cost ($) | Model | Mass (kg) |
|---|---|---|---|
| ECU_3 | 155.00 | Arty A7-35T | 0.883 |
| ECU_5 | 198.00 | AX7Z010B | 1.000 |
| ECU_7 | 213.00 | AX7Z010B | 1.000 |
| ECU_8 | 140.00 | Arty A7-35T | 0.883 |
| ECU_10 | 198.00 | AX7Z010B | 1.000 |
| BUS_18 | 0.25 | CAN | 0.045 |
| BUS_36 | 0.20 | LIN | 0.025 |
| BUS_38 | 0.25 | CAN | 0.045 |
| BUS_48 | 0.25 | CAN | 0.045 |
| **Total** | **904.95** | – | **4.926** |

Table 3.13: ECU costs, models, and masses (Multi-objective max dim 12 Configuration 7)

The same procedure was carried out for the case with a maximum of 15 functions per ECU. The results are summarized in Figure 3.39, which presents the complete Pareto front, and Figure 3.40, which provides a zoomed view of the most relevant region of the solution space. Table 3.14 reports the feasible configurations that compose the Pareto-optimal set, from which two representative solutions have been selected for closer analysis. Figure 3.41 illustrates the configuration achieving a trade-off solution, while Figure 3.42 shows the configuration with the minimum weight. As in the previous case, these examples highlight how different optimization priorities can lead to distinct but valid ECU network architectures. The comparison between the two solutions demonstrates the nature of the trade-offs: the weight-minimized configuration reduces system mass at the expense of higher costs, whereas the balanced configuration provides significant economic savings while requiring a slightly heavier system. This dual perspective once again emphasizes the value of multi-objective optimization, which enables decision-makers to balance competing criteria and select the architecture that best aligns with project goals.

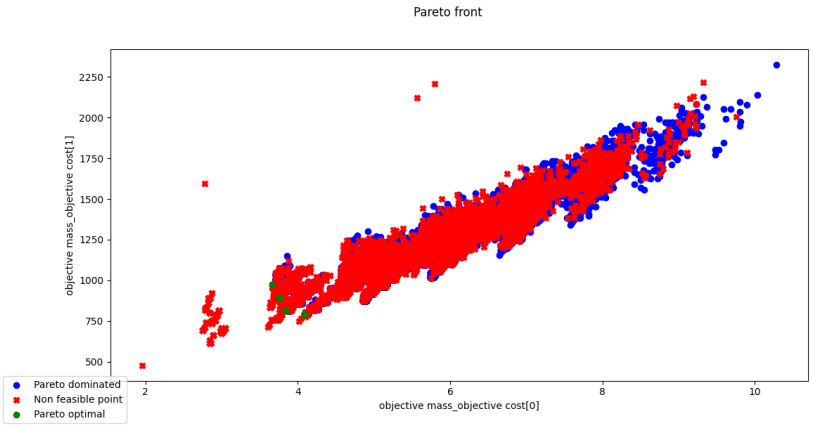


Figure 3.39: Pareto front of the multi-objective optimization continuous database max dim 15

| Configuration | Cost[$] | Mass [Kg] |
|---|---|---|
| 1 | 971.25 | 3.667 |
| 2 | 892.25 | 3.754 |
| 3 | 892.25 | 3.754 |
| 4 | 813.25 | 3.841 |
| 5 | 792.5 | 4.09 |

Table 3.14: Pareto-optimal configuration (max dim 15)

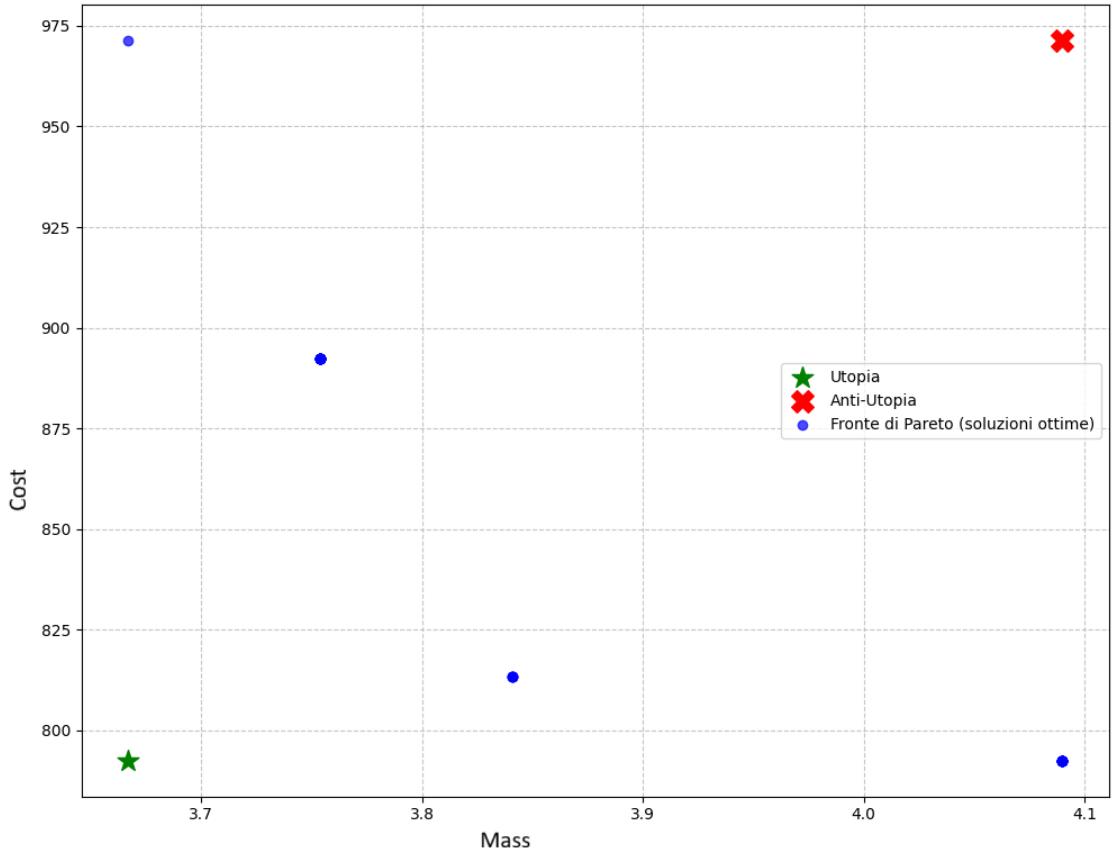


Figure 3.40: Zoom of pareto-front of the multi-objective optimization continuous database max dim 15

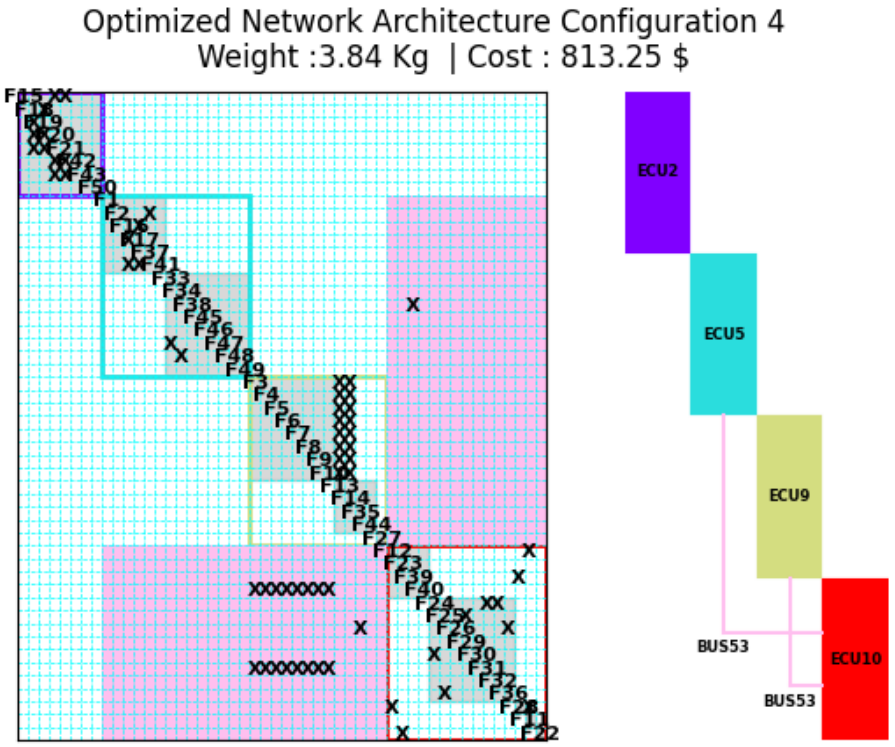


Figure 3.41: Multi-objective Optimized Network Architecture Configuration 4 (max dim 15)

| **ECU** | **Cost ($)** | **Model** | **Mass (kg)** |
|---|---|---|---|
| ECU_2 | 140.00 | Arty A7-35T | 0.883 |
| ECU_5 | 198.00 | AX7Z010B | 1.000 |
| ECU_9 | 277.00 | AX7Z010B | 1.000 |
| ECU_10 | 277.00 | Arty A7-100T | 0.913 |
| BUS_53 | 0.25 | CAN | 0.045 |
| **Total** | **813.25** | – | **3.841** |

Table 3.15: ECU costs, models, and masses (Multi-objective max dim 15 Configuration 4)

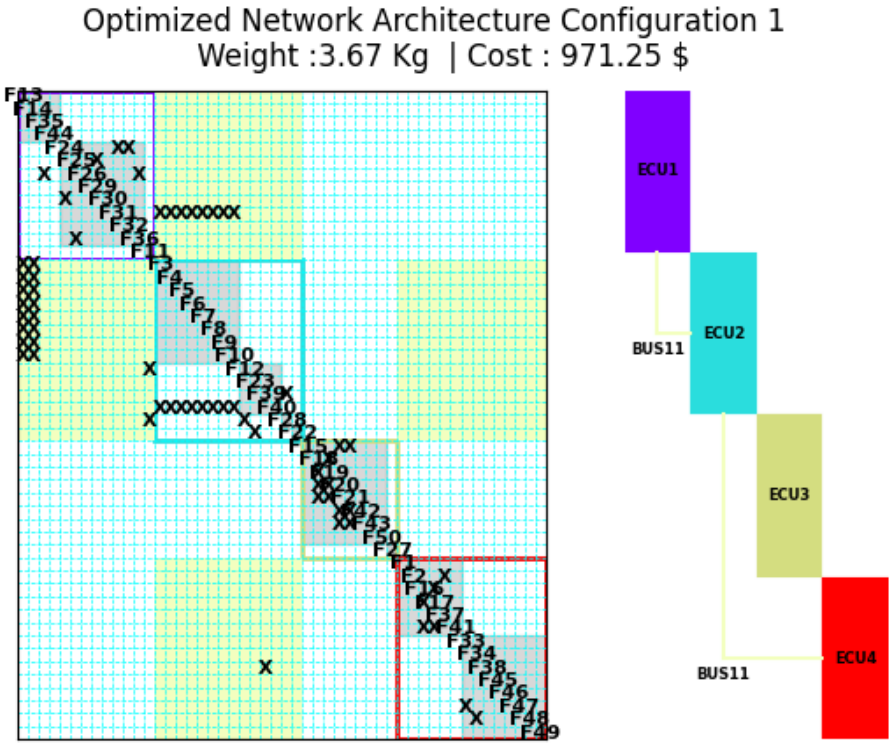


Figure 3.42: Multi-objective Optimized Network Architecture Configuration 1 (max dim 15)

| ECU | Cost ($) | Model | Mass (kg) |
|---|---|---|---|
| ECU_1 | 277.00 | Arty A7-100T | 0.913 |
| ECU_2 | 277.00 | Arty A7-100T | 0.913 |
| ECU_3 | 140.00 | Arty A7-35T | 0.883 |
| ECU_4 | 277.00 | Arty A7-100T | 0.913 |
| BUS_11 | 0.25 | CAN | 0.045 |
| **Total** | **971.25** | – | **3.667** |

Table 3.16: ECU costs, models, and masses (Multi-objective max dim 15 Configuration 1)

# Conclusions

This thesis has addressed the challenge of designing and optimizing automotive electronic architectures by integrating MBSE and MDO into a unified framework. Motivated by the increasing complexity of modern vehicles, where the number of Electronic Control Units has grown significantly due to rising functionality and connectivity demands, the work has sought to demonstrate how digital continuity between system modeling and optimization can support early and informed decision making. The primary contribution of this thesis lies in the integration of MBSE and MDO. SysML models were developed in Cameo Systems Modeler, where requirements, functions, and architectures were captured in a traceable structure to ensure consistency across different system views; from these models, structured CSV tables were extracted, representing the interactions among functions, computational loads, and communication signals. These datasets served as the direct inputs to the Python optimization environment, which was implemented using the GEMSEO framework. This workflow represents digital continuity, permitting the easy transition from modeling the system level to optimizing quantitatively. The second contribution involves the processes of developing and applying optimization strategies. The application of optimization strategies entailed consideration of both single-objective optimization strategies and multi-objective optimization strategies while using both continuous curve-fitted models and discrete catalog-based ECU databases. Constraints such as maximum computational load, number of interfaces, and functional group allocations were explicitly included to ensure the feasibility of the optimized architectures. The results also illustrate clear trends in these areas; single-objective optimization identified the relationship between the number and complexity of ECUs and system cost, while multi-objective optimization produced Pareto fronts for the trade-off of cost and weight. Although integrating MBSE with MDO has shown its worth and provided some good results, the analysis also points out a number of limitations of the current framework. For example, the comparison between the continuous cost model and the discrete database shows a significant divergence in results, indicating that the polynomial fitting process used in the continuous model could not fully capture the real cost–performance behavior of ECUs. A broader and more representative database would reduce this gap, but its creation proved difficult due to the limited availability of technical and economic data, much of which remains confidential within manufacturers. Another limitation concerns the estimation of function weights: instead of measuring the actual computational demand of each function, the loads were approximated by categorizing them, which inevitably introduces simplifications into the model. Even with these obstacles, the result obtained from the multi-objective optimization effort are quite positive; it shows the capacity of the framework to optimize costs and weights concurrently, while suggesting Pareto-optimal configurations that offer reasonable trade-offs for system architects. This validates the added value of MBSE and MDO to support early stage decision making and to establish alternative architectures within real-system-level constraints. Onward into the future, multiple paths for future work present themselves. First, developing the ECU database and developing a real data set for communication buses would enable dramatic improvement of the optimization process and fidelity of results. Second, applying continuous model to multi-objective optimization would allow a more comprehensive investigation

of the design space and smoother trade-off curves between competing objectives. Third, reintegrating optimized ECU networks directly into Cameo would be a significant step toward achieving a complete digital continuity and would further close the loop between system modeling and optimization. In addition to these improvements, the incorporation of additional disciplines into the optimization framework is encouraged. For example, adding power consumption as an optimization variable would allow consideration of energy efficiency alongside cost and weight, which is particularly relevant for electric and hybrid vehicles. In addition, considering the placement of ECUs within the vehicle would provide spatial dimensions to the problem and incorporate packaging constraints, wire lengths, and thermal management. In conclusion, this thesis illustrates the strong potential of integrating system modeling with optimization to address the challenges posed by increasingly complex automotive electronics. The framework developed here improve traceability, decision-making, and establishes the basis for conducting comprehensive multidisciplinary trade-off analyses. Beyond the immediate results, it also lays a solid foundation for future extensions and industrial applications, pointing toward a more efficient, robust, and systematic approach to the design of next-generation automotive architectures.

# Bibliography


[1] TechWorld002, "Main components and systems of motor." https://techworld002.blogspot.com/2016/06/main-components-and-systems-of-motor.html, 2016.

[2] E. Brusa, A.Calà, and D. Ferretto, *Systems Engineering and Its Application to Industrial Product Development*. Springer, 2017.

[3] A. M. Madni and S. Purohit, "Economic analysis of model-based systems engineering," *Systems*, vol. 7, no. 1, pp. 1–12, 2019.

[4] Sandia National Laboratories, "Mbse at snl." https://www.sandia.gov/digital-engineering/new-to-mbse/mbse-at-snl/, 2025.

[5] M. Bouaicha, N. Machkour, I. E. Adraoui, and M. Zegrari, "Mbse grid: Operational analysis for the implementation of hydroelectric group health monitoring and management unit," in *Smart Applications and Data Analysis (SADASC 2022)*, Communications in Computer and Information Science, Springer, Cham, 2022.

[6] K. Hardman, "What is systems design? part 3: The case for model based systems engineering." https://www.designreview.byu.edu/collections/what-is-systems-design-part-3-the-case-for-model-based-systems-engineering, 2023. Brigham Young University Design Review.

[7] H. Shahid, "Integration of system-level design and mechanical design models in the development of mechanical systems," master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2011. Supervisor: Ahsan Qamar.

[8] S. C. Spangelo, J. W. Cutler, L. Anderson, E. Fosse, L. Cheng, R. Yntema, M. Bajaj, C. Delp, B. Cole, and D. Kaslow, "Model based systems engineering (mbse) applied to radio aurora explorer (rax) cubesat mission operational scenarios," in *2013 IEEE Aerospace Conference*, (Big Sky, MT, USA), pp. 1–8, IEEE, Mar. 2013.

[9] J. Martins, "Multidisciplinary design optimization: An introduction," in *MDO Course Slides*, University of Michigan, 2016. Lecture slides.

[10] J. R. R. A. Martins and A. B. Lambe, "Multidisciplinary design optimization: A survey of architectures," *AIAA Journal*, vol. 51, pp. 2049–2075, Sept. 2013.

[11] Quattroruote, "Centralina elettronica di gestione motore: cos'è, a cosa serve e come funziona." https://www.quattroruote.it/guide/componenti-auto/centralina-gestione-motore.html, 2025.

[12] M. Mandorino, *Model-Based Systems Engineering for Multidisciplinary Optimization of Advanced Aircraft Rear-Ends*. PhD thesis, Università degli studi di Napoli Federico II, 2024.

[13] INCOSE, *INCOSE Systems Engineering Vision 2025*. San Diego, CA, USA: International Council on Systems Engineering, 2014. Available at: https://www.incose.

org/sevision2025.

[14] J. A. Estefan, "Survey of candidate model-based systems engineering (mbse) methodologies, rev. b," Tech. Rep. INCOSE-TD-2007-003-02, International Council on Systems Engineering (INCOSE), Seattle, WA, USA, 2008. http://www.omgsysml.org/MBSE_Methodology_Survey_RevB.pdf.

[15] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*. Addison-Wesley, 2014.

[16] D. Dori, *Model-Based Systems Engineering with OPM and SysML*. Springer, 2016.

[17] International Organization for Standardization, "Iso/iec/ieee 15288:2023 - systems and software engineering — system life cycle processes," 2023. ISO/IEC/IEEE Standard.

[18] J. R. R. A. Martins and A. Ning, *Engineering Design Optimization*. Cambridge University Press, 2021.

[19] INCOSE, *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. Wiley, 4th ed., 2015.

[20] SEBoK Editorial Board, "The guide to the systems engineering body of knowledge (sebok)." {https://www.sebokwiki.org}„ 2024.

[21] Georgia Institute of Technology Professional Education, "Model-based systems engineering fundamentals," 2025. Available: https://pe.gatech.edu/blog/industry-trends/model-based-systems-engineering-fundamentals.

[22] S. D. Ross, A. Kossiakoff, W. N. Sweet, and S. J. Seymour, *Systems Engineering: Principles and Practice*. Wiley, 2nd ed., 2014.

[23] W. W. Royce, "Managing the development of large software systems," in *Proceedings of IEEE WESCON*, 1970.

[24] B. W. Boehm, "A spiral model of software development and enhancement," in *ACM SIGSOFT Software Engineering Notes*, vol. 11, ACM, 1988.

[25] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*. Waltham, MA: Morgan Kaufmann, 3rd ed., 2015.

[26] C. E. Dickerson and D. Mavris, "A brief history of models and model based systems engineering and the case for relational orientation," 2013. Manuscript, revised March 4, 2013. Available via Loughborough University Repository.

[27] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Burlington, MA: Morgan Kaufmann, 2nd ed., 2011.

[28] J.-C. Chaudemar and P. de Saqui-Sannes, "Mbse and mdao for early validation of design decisions: a bibliography survey," (Virtual Conference, Canada), 04 2021. 10.1109/SysCon48628.2021.9447140.

[29] Embitel Technologies, *A Handbook of Automotive Electronic Control Units (ECU)*. Embitel Technologies, Bangalore, India, 2018. https://www.embitel.com/wp-content/uploads/pdf/ECU-Handbook.pdf.

[30] S. Lundh, "Programming of automotive electronic control units over can bus," thesis in computer science, Linnaeus University, Faculty of Technology, Department of Computer Science and Media Technology (CM), Växjö, Sweden, 2022.

[31] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Elsevier/Newnes, 2004.

[32] Intel Corporation, *Altera Product Catalog*, 2023. Available at: http://www.intel.com/fpga.

[33] Intel Corporation, *Automotive Brochure*, 2015. Available at: https://www.intel.com/content/www/us/en/automotive/products/programmable/applications.html.

[34] Intel Corporation, *Nios® V Processor Reference Manual*, 2023. Available at: https://www.intel.com/content/www/us/en/docs/programmable/683082/current/nios-v-processor-reference-manual.html.

[35] I. AMD (Advanced Micro Devices, "Microblaze processor." https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/microblaze.html„ 2025.