



**Politecnico  
di Torino**

**Politecnico di Torino**

Master's Degree in Aerospace Engineering

A.Y. 2024/2025

Graduation Session July 2025

# **Integrating Data-Driven and Model-Based Systems Engineering for End-to-End Small-Satellite Design**

**Academic Supervisors:**

Prof. Fabrizio Stesina

Prof. Sabrina Corpino

PhD(c). Serena Campioli

**Candidate:**

Carlotta Deiana



# Summary

The aerospace industry is currently experiencing a significant transition towards the digitalisation of spacecraft development processes. This shift presents several challenges, including technical limitations in data exchange systems and the increasing complexity of space systems. Over the years, a variety of System Engineering (SE) approaches have been adopted with the objective of managing information as it passes between engineering teams, ensuring consistent design data across different projects and activities, and reducing mission costs and development times. The most relevant of these approaches are Data-Driven Systems Engineering (DDSE) and Model-Based Systems Engineering (MBSE). The DDSE centralises and integrates all space mission engineering data, creating a single source of truth for rapid and collaborative decision-making and enabling faster automation and iteration. The MBSE employs a model-centric approach to the design of complex space systems, promoting a coherent understanding and early validation of mission architectures.

To address the challenges currently being experienced in the aerospace industry, the design of small satellites requires a methodology that is more agile than traditional approaches. Moreover, a single modelling tool is often inadequate. This thesis proposes an integration of the Data-Driven System Engineering (DDSE) and Model-Based System Engineering (MBSE) methodologies with the aim of enhancing and facilitating the preliminary small satellite design development. To achieve this objective, two complementary tools have been selected: Valispace, a DDSE tool that provides a user-friendly digital environment for engineering data, ensuring the integrity of data while encouraging real-time collaboration; Capella, an MBSE tool that implements the ARCADIA method to create, visualize and simulate the models of the systems, improving the comprehension of the project even at an early stage.

The integrated approach is based on the combination of these two methodologies, thereby ensuring a more comprehensive and clear understanding of the space system

being modelled, and achieving a more efficient and dynamic management of all data. This, in turn, allows for the maintenance of a single source of truth, guaranteeing the continuity of information between the two tools and fostering the collaboration between engineering teams and stakeholders. The present thesis illustrates the methodology, the integration of the tools and the results from a small-satellite case study. It demonstrates how the combination of DDSE and MBSE has optimised the design development process, thereby ensuring a more agile solution that mitigates risks and improves decision-making processes.





# Table of Contents

<b>List of Figures</b>	VIII
<b>List of Tables</b>	XI
<b>List of abbreviations</b>	XII
<b>Introduction</b>	1
<b>1 State of Art</b>	3
1.1 Systems Engineering . . . . .	3
1.2 Data-Driven Systems Engineering . . . . .	5
1.3 Model-Based Systems Engineering . . . . .	8
1.3.1 Methodologies . . . . .	10
1.3.2 Modelling languages . . . . .	13
<b>2 Systems Engineering Tools</b>	15
2.1 Valispace . . . . .	17
2.2 Capella . . . . .	20
<b>3 Valispace-Capella Integration</b>	27
3.1 DDSE-MBSE approach . . . . .	27
3.2 Integration . . . . .	30

Stakeholders Needs Analysis . . . . .	32
System Analysis . . . . .	32
System Design . . . . .	33
Subsystems Design . . . . .	34
3.3 <i>Python for Capella</i> . . . . .	34
<i>Export_SF_MO.py</i> . . . . .	36
<i>Export_System_Actor.py</i> . . . . .	36
<i>Export_Logical_Component.py</i> . . . . .	36
<i>Export_modes.py</i> . . . . .	37
3.4 Valispace Python API . . . . .	37
<i>Import_Requirements.py</i> . . . . .	37
<i>Import_System_Components.py</i> . . . . .	38
<i>Import_LC_Hierarchy.py</i> . . . . .	38
<i>Components_bridge.py</i> . . . . .	38
<i>Import_All_Modelists.py</i> . . . . .	39
<b>4 Case study</b>	<b>40</b>
4.1 Mission Overview . . . . .	40
4.2 Stakeholders Needs Analysis . . . . .	42
4.3 System Analysis . . . . .	49
4.4 System Design . . . . .	59
4.5 Subsystem Design . . . . .	69
<b>Conclusions</b>	<b>72</b>
<b>A <i>Python for Capella</i> scripts</b>	<b>75</b>
A.1 <i>Export_SF_MO.py</i> . . . . .	75
A.2 <i>Export_System_Actor.py</i> . . . . .	78
A.3 <i>Export_Logical_Component.py</i> . . . . .	79

A.4	<i>Export_modes.py</i>	81
<b>B</b>	<b>Valispace Python API scripts</b>	<b>83</b>
B.1	<i>Import_Requirements.py</i>	83
B.2	<i>Import_System_Components.py</i>	85
B.3	<i>Import_LC_Hierarchy.py</i>	86
B.4	<i>Components_bridge.py</i>	88
B.5	<i>Import_All_Modelists.py</i>	95
<b>C</b>	<b>Capella diagrams</b>	<b>97</b>
	<b>Bibliography</b>	<b>102</b>

# List of Figures

1.1	The engineering data pyramid [5]. . . . .	7
1.2	The PMTE Elements and Effects of Technology and People. [3] . .	11
2.1	Valispace V-model [18]. . . . .	18
2.2	Valispace overview. . . . .	19
2.3	The main engineering levels of the Architecture Analysis & Design Integrated Approach (ARCADIA) Method [17]. . . . .	21
3.1	Tools integration process. . . . .	31
3.2	<i>Python for Capella</i> architecture [22]. . . . .	35
4.1	Operation Capabilities Blank (OCB) Operational Capabilities . . .	44
4.2	Operational Activity Interaction Blank (OAIB) Mission Lifecycle . .	45
4.3	OAIB [Scientific Obj] Urban Heat Island (UHI) effect mapping and detection . . . . .	45
4.4	OAIB [Technological Obj] Deep space technologies validation in Low Earth Orbit (LEO) . . . . .	46
4.5	Operational Architecture Blank (OAB) Operational Entities . . . .	47
4.6	Operational Scenario (OES) Mission Lifecycle . . . . .	48
4.7	Context System Actors (CSA) System . . . . .	50
4.8	System Components imported from Capella to Valispace. . . . .	51
4.9	Mission and Capabilities Blank (MCB) Mission and Capabilities Blank diagram . . . . .	52

4.10	System Functions Breakdown Diagram (SFBD) To maintain system operability . . . . .	52
4.11	SFBD To satisfy mission objectives . . . . .	53
4.12	Mission Requirements imported from Capella to Valispace. . . . .	54
4.13	System Data Flow Blank (SDFB) To reach the operational orbit . .	55
4.14	SDFB Data flow . . . . .	55
4.15	ConOps visual. . . . .	56
4.16	System Architecture Blank (SAB) Structure . . . . .	57
4.17	SAB Operative phase . . . . .	58
4.18	Logical Functions Breakdown Diagrams (LFBD) To satisfy mission objectives . . . . .	60
4.19	LFBD To maintain system operability . . . . .	61
4.20	Logical Component Breakdown Diagram (LCBD) Space Segment .	62
4.21	Logical Component imported from Capella to Valispace. . . . .	63
4.22	<b>Component_bridge.py</b> output in console when Thermal Control System (TCS) component is missing in Valispace. . . . .	63
4.23	Logical Architecture Blank (LAB) Structure . . . . .	64
4.24	Modes and State Diagrams (MSM) Operative modes . . . . .	66
4.25	Operative modes imported into Valispace from Capella . . . . .	67
4.26	"Power" property associated with Attitude and Orbit Control System (AOCS)&Guidance, Navigation and Control (GNC) component, depending on the operative modes. . . . .	68
4.27	Power budget estimation on Valispace for <b>In-orbit check-out</b> mode.	68
4.28	Physical Architecture Blank (PAB) Structure Electrical Power System (EPS) . . . . .	70
4.29	Battery trade-off on Valispace. [32] [33] . . . . .	71
C.1	SFBD To position the system . . . . .	97
C.2	SFBD To perform ground operations . . . . .	97
C.3	SFBD To implement disposal operations . . . . .	97

C.4	SAB Integration & Testing (I&T)	98
C.5	SAB Launch and Early Orbit Phase (LEOP)	99
C.6	SAB End Of Life (EOL)	100
C.7	Mass budget estimation on Valispace.	101
C.8	Volume budget estimation on Valispace.	101

# List of Tables

1.1	Strenghts and weaknesses of Data-Driven Systems Engineering (DDSE)	7
1.2	Strenghts and weaknesses of Model-Based Systems Engineering (MBSE)	10
2.1	Overview of types of diagrams available in Capella [17].	24
3.1	DDSE and MBSE integration.	28
3.2	Valispace and Capella capabilities [20].	29
4.1	High Level Requirements and Drivers	41
4.2	Operational Analysis terminology [19].	42
4.3	System Analysis terminology [19].	49
4.4	Logical Architecture terminology [19].	59
4.5	Physical Architecture terminology [19].	69



# List of the abbreviations

**AIV/T** Assembly, Integration, Verification and Testing

**AOCS** Attitude and Orbit Control System

**ARCADIA** Architecture Analysis & Design Integrated Approach

**API** Application Programming Interface

**AutomationML** Automation Modeling Language

**CAM** Collision Avoidance Manoeuvre

**ConOps** Concept of Operations

**COTS** Commercial-Off-The-Shelf

**CRUD** Create-Read-Update-Delete

**CSA** Context System Actors

**DDSE** Data-Driven Systems Engineering

**DSL** Domain-Specific Language

**EO** Earth Observation

**EOL** End Of Life

**EPBS** End Product Breakdown Structure

**EPS** Electrical Power System

**ESA** European Space Agency

**FDIR** Failure Detection, Isolation and Recovery

**GNC** Guidance, Navigation and Control

**HTTP** Hypertext Transfer Protocol

**ID** Identification

**IDE** Integrated Development Environment

**IML** Interdisciplinary Modeling Language

**INCOSE** International Council on Systems Engineering

**IVVQ** Integration, Verification, Validation, and Qualification

**I&T** Integration & Testing

**JPL** Jet Propulsion Laboratory

**LAB** Logical Architecture Blank

**LCBD** Logical Component Breakdown Diagram

**LEO** Low Earth Orbit

**LEOP** Launch and Early Orbit Phase

**LFBD** Logical Functions Breakdown Diagrams

**LST** Land Surface Temperature

**MBSE** Model-Based Systems Engineering

**MCB** Mission and Capabilities Blank

**MCC** Mission Control Center

**MDD** Model-Driven Development

**MDS** Model-Driven System

**MNDWI** Modified Normalised Difference Water Index

**MSM** Modes and State Diagrams

**NDBI** Normalised Difference Built-up Index

**NDVI** Normalised Difference Vegetation Index

**OA** Operationa Analysis

**OAB** Operational Architecture Blank

**OAIB** Operational Activity Interaction Blank

**OCB** Operation Capabilities Blank

**OES** Operational Scenario

**OOSEM** Object-Oriented Systems Engineering Methodology

**OPD** Object-Process Diagram

**OPL** Object-Process Language

**OPM** Object-Process Methodology

**PAB** Physical Architecture Blank

**PCDU** Power Control and Distribution Unit

**PCU** Power Control Unit

**PDU** Power Distribution Unit

**PI** Principal Investigator

**REST** Representational State Transfer

**RUP** Rational Unified Process

**RUP-SE** Rational Unified Process for System Engineering

**s/c** Spacecraft

**SA** System Analysis

**SAB** System Architecture Blank

**SDFB** System Data Flow Blank

**SDL** System Definition Language

**SE** Systems Engineering

**SFBD** System Functions Breakdown Diagram

**SYSMOD** Systems Modeling Process

**SoI** System of Interest

**SoW** Statement of Work

**SQL** Structured Query Language

**SysML** Systems Modeling Language

**TCS** Thermal Control System

**TRL** Technology Readiness Level

**UHI** Urban Heat Island

**UML** Unified Modeling Language

**VPMS** ViewPoint Modes and States

# Introduction

The present thesis is contextualised within the field of Systems Engineering (SE), with a specific focus on the methodologies employed in the aerospace industry for small satellite missions.

Among these methodologies, the ones that have emerged most recently are DDSE and MBSE. The DDSE approach promises centralised management of quantitative and qualitative engineering data, ensuring consistency throughout the entire product life cycle. In contrast, MBSE adopts a model-centric approach to modelling system architecture and functionalities, guided by the ARCADIA method.

However, a more exhaustive methodology is required, capable of combining the strengths of both approaches with the aim of enhancing the design process of small satellites. To achieve this objective, two tools have been selected: Valispace and Capella, which represent the DDSE and MBSE methodologies, respectively.

The proposed solution in this thesis is based on the exploitation of the advantages of each approach shown in each respective tool, whilst simultaneously covering each other's weaknesses through the integration of them. The developed integration acts as a bridge between the two environments, with the objective of facilitating the transfer of information between them while ensuring the retention of a single source of truth. Consequently, each tool will be responsible for managing the project activities for which it is optimally suited, whilst also acquiring information from the other tools in order to ensure a consistent and constantly updated workflow. The execution of this process is supported by the utilisation of the Application Programming Interface (API) of the two software programs.

In summary, the thesis deals with the integration of these two methodologies for the development of a small satellite mission and is structured as follows.

The initial chapter presents a thorough overview of the thesis's background, delving into the domain of SE, with a particular emphasis on the two methodologies employed: DDSE and MBSE.

The second chapter conducts a meticulous investigation into the tools that have been selected, Valispace and Capella, including a detailed examination of their

potentialities, strengths and weaknesses.

The third chapter of this work of thesis focuses on the proposed solution, the integration between the two tools, and provides a detailed explanation of its application in the early stages of the mission, with a particular focus on the Python codes that made it possible.

The fourth chapter presents a case study, consisting in a small satellite mission, called GeoProfundo, with dual objectives: the emulation of an interplanetary environment and Earth observation.

Finally, a concise section on conclusions summarises the outcomes of this work and provides a prospectus for future research related to the present thesis.

# Chapter 1

## State of Art

### 1.1 Systems Engineering

The SE multi-disciplinary and methodical [1] approach is described by the International Council on Systems Engineering (INCOSE) as "*a means to enable the realization of successful systems*" and "*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*" [2]. SE can be defined as the art of developing a functioning system capable of both satisfying requirements and observing the constraints imposed on it. Its objective is to enhance the project as a whole, balancing operational, technical, economic, and logistical factors, and producing a product that is not dominated by the perspective of a single discipline. It is a holistic and integrative discipline in which engineers with different backgrounds work in collaboration to produce a coherent outcome [1] [3].

Indeed, a *system* can be defined as a collection of different elements that, when considered as a whole, produce results that would not be obtainable by the elements alone. The elements may involve hardware, software, equipment, facilities, personnel, and procedures that are necessary for the purpose in question. In summary, these are all the elements that are indispensable for the generation of system-level results. The quality of the system is primarily determined by the interaction and interconnection among its parts, with the aim of improving global system performance. [1]

SE is a "*logical way of thinking*" [1], a unique way of viewing reality, with the capacity to increase awareness of the entire system and the way in which



its components interact. The SE process is characterised by an iterative methodology, which supports discovery, learning and continuous improvement. It combines technical and management processes, both of which depend on effective decision-making. [1]

In the contemporary era of SE, it is crucial to address several pivotal challenges. Firstly, **handling complexity** is of primary importance, given that current products are complex systems composed of subsystems, components, and parts, frequently equipped with smart functionalities and communication capabilities. Conventional tools are no longer sufficient to manage the increasing complexity of systems. [4]

Secondly, it is essential to ensure **traceability** across the entire life cycle. In the event of inadequate traceability during the development of products, costs for the manufacturer can increase rapidly.

Furthermore, **model creation, digitalisation, reusability, and automatic documentation** [4] are vital aspects. In order to ensure that users and operators are fully informed about system operation, maintenance, and even failures, it is necessary to produce comprehensive documentation that is both complete and exhaustive. This approach serves to reinforce the manufacturer's expertise and provides a framework for enhanced user support. In contrast to static documents, *models* offer significant advantages in terms of reusability, sharing, and storage for automated documentation production.

Finally, SE aims to **reduce system development costs, human mistakes, and late re-engineering activities** [4]. The growing prevalence of SE within the industrial sector can be attributed to its efficacy in managing multidisciplinary engineering projects.

As demonstrated in the literature [4], there are four key areas of interest in SE: (i) **the methodology**, based on *digital models*, which are easily shared within a community of users and stored by a data management system; (ii) **the tools**, encompassing both theoretical tools, which include several typical diagrams and certain engineering methods, and software that provides a digital and virtual environment managed by a standard language; (iii) **the language**, establishing a common terminology or a tool that is easily understood and applied by all developers is necessary for promoting efficient cooperation; (iv) **the data management**, that requires a common environment with authorised user access, including hardware and software. The platform enables interoperability, automatically transferring data without user intervention.

The fundamental principles of SE have been delineated. It is evident that this field is undergoing a constant evolution to address increasingly exigent

challenges, especially those driven by the growing complexity of systems and the necessity for enhanced integration between the physical and digital domains. In this context, two modern methodologies are transforming the approach to SE: DDSE and MBSE. These two concepts will be analysed more in detail in the following sections.

## 1.2 Data-Driven Systems Engineering

DDSE is a "*methodology where engineering data and associated structure, links and connections constitute the foundation of the systems engineering process*" [5]. The DDSE is founded on the principle of data collection in a unified database, with concurrent access for all engineers engaged in the project. The main objective is the establishment of a centralised location for data management, which is constantly updated and monitored. This database must have the capacity to immediately detect and respond to changes while also maintaining a comprehensive overview of the relationships between values. [5]

While document-centric approaches involve the creation of documents to report on the current state at the conclusion of each development phase, DDSE focuses on working with data from a single source of truth. The concept of a *single source of truth* implies that all data points are created, utilised and stored within a unified database. The engineers benefit from total oversight of the project data and its latest changes, facilitating the ability to make optimal decisions and trade-offs with all the relevant information in real time. While the document-centric approach has historically been an adopted methodology in engineering activities, a data-driven approach offers the potential to minimise inconsistencies in engineering data and the time expended on document updates and information acquisition. [6]

The majority of companies in the space industry continue to utilise a document-driven approach to engineering. In traditional SE, the outcome of each design stage is a set of documents that outline deliverables of the activity that serve as input for the succeeding stage in the development process. The documents are then often subjected to further modifications, which serve to increase the level of design of the product. This process can introduce further dependencies and technical changes, which, in turn, may result in document inconsistencies. The necessity for accurate and effective data management is increasing significantly, and the DDSE allows for this consistency. [5]

The DDSE approach offers a wide range of benefits. The creation of a

**consistent database** of connected engineering values is of crucial importance, not only for DDSE, but also for SE as a whole. This leads to the establishment of accurate assumptions and analyses at the early stages of the project. The identification of discrepancies during the latter stages of a project can indeed be a costly and time-consuming process, often necessitating additional expenses and rework. Furthermore, it is crucial that engineering values are interconnected through the utilisation of formulas or simulations, thereby facilitating automatic propagation that ensures consistency throughout the entire development life cycle. Another feature pertains to **automation**. The use of scripts to execute manual operations has been demonstrated to enhance efficiency and to reduce the probability of errors. APIs support the integration of tools and the exchange of data, thereby establishing connections between different disciplines and areas of research.

In the context of **traceability and transparency**, DDSE exhibits multi-user functionality and facilitates clear interconnections between data and users, thereby transforming data into a dynamic entity. Traceability is the ability to identify the value derived at any moment by anyone on a project.

In conclusion, DDSE facilitates the implementation of **optimisation** processes, thereby empowering engineers to exploit the modelled relationships and enable the automatic derivation of optimised versions of the project. [5]

However, this methodology has also its limitations. Incomplete or incorrect data can compromise design choices. It is evident that any analysis or model that is constructed on the basis of these elements will inevitably produce incorrect results and lead to erroneous decisions. Moreover, the significant reliance on data leads to increased costs. The implementation of a DDSE infrastructure necessitates a substantial initial investment, in addition to continuous maintenance expenses. Data privacy and security represent another key concern. Complex systems frequently manage sensitive and private data and are thus required to comply with regulations.

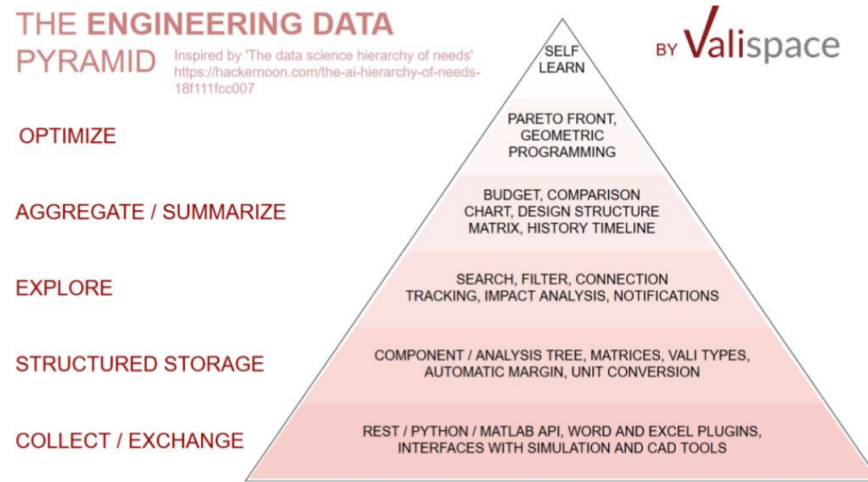
Finally, it is important to highlight the DDSE's capacity to generate large volumes of data, as this can delay decision-making and project organisation, thus increasing development costs and times.

The following table 1.1 provides a summary of the advantages and disadvantages of this methodology.

**Table 1.1:** Strenghts and weaknesses of DDSE

DDSE features	
Strenghts	Weaknesses
Consistent database of engineering data	Critical dependence on data
Automation through APIs	High costs on infrastructure
Traceability and transparency	Data privacy and security
Constant project optimization	Slow decision-making process
Expert personnel not required	

In order to implement DDSE, it is first necessary to establish an infrastructure that permits simple instrumentation and data access for all relevant stakeholders. The processes of data collection and storage are at the foundation of an engineering data management system, as shown in figure 1.1.

**Figure 1.1:** The engineering data pyramid [5].

In the domain of engineering data management, the "*hierarchy of data needs*" states that certain fundamental elements of data storage and structure are prerequisites for the effective execution of data analytics, optimisation algorithms and machine learning. For this reason, after the data collection into a unified database, the subsequent stage of the process involves the integration of a structured framework within the data. In order to facilitate clear communication and analysis, values are structured using a simple data model, thus enhancing collaboration and greater

transparency. Finally, the levels at the top of the pyramid allow advanced exploration, powerful data analytics and automation. [5]

In order to fully embrace a data-driven approach, it is important to establish connections between tools and systems across various disciplines through the utilisation of open APIs. The assurance of consistency in the final design of a space project is dependent on the effective exchange of data between various engineering tasks and their corresponding tools during the design phases. [5]

### 1.3 Model-Based Systems Engineering

Nowadays, the increasing complexity of systems has led to the evolution of concepts and to the research of innovative approaches to implement SE. MBSE has emerged as a strong methodology to support the management of complexity, the maintenance of consistency, and to assure traceability during system development. It is a SE approach defined by the INCOSE as "*the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases*" [2].

The MBSE methodology is a collection of processes, methods and tools, both theoretical and software, that support the discipline of SE within a "model-based" context. *Models* are considered the fundamental technical baseline for a project, with the purpose of supporting and facilitating the performance of the systems engineering tasks throughout the system's life cycle. These models, which can be physical, mathematical, or otherwise logical representations of a system, entity, phenomenon or process, are utilised from requirements and analysis definition, through design and implementation, and finally to the verification of the system. A model is not merely a collection of representations, but an integrated repository of the project's knowledge. [7] [8]

MBSE stands as a pillar of the digital transformation in SE. It provides a "*more holistic, collaborative, and efficient approach*" [9] to develop complex systems. In the document-based method, system information and specifications are primarily generated as documents. The huge number of documents often leads to significant management and updating challenges. It becomes crucial to prevent different team members from simultaneously working on conflicting versions of the same document, thereby mitigating communication issues. Model-Based Systems Engineering, in contrast, formalizes systems engineering practices, utilizing coherent digital system and engineering domain models as the primary means of exchanging information, feedback, and requirements. [9]

This approach has been demonstrated to offer significant advantages. These include an **increased productivity** through the reuse of existing models and automated generation of documentation, thus reducing both time and costs.

Furthermore, this approach **enhances communication**, thereby optimising the exchange of information and establishing a common language not only among engineers from diverse backgrounds, but also between designers and clients.

In addition, the methodology **improves the quality** of the system through the establishment of a unique, authoritative system model that supports rigorous requirements traceability, enhanced system design integrity and consistent documentation.

MBSE also offers a significant improvement in **complexity management**, facilitating the effective handling of intricate systems by decomposing them into more manageable components. This, in turn, enables the effective mitigation of risk.

It is finally characterised by its ability to facilitate **early defect detection**, thereby minimising inconsistencies and errors, to achieve a more robust and reliable final product. The validation of requirements through design verification serves to increase reliability and reduce risks. [10] [11]

Nevertheless, this approach has its drawbacks. Initially, the adoption of this approach is not an immediate, but rather a gradual process, that necessitates substantial change and training for the team. This involves the acquisition of new instruments and the assimilation of novel modelling languages. It is not unexpected that teams familiarised with more user-friendly, document-based methods may be resistant to such a transition.

Financially, MBSE requires an initial investment to procure the necessary tools for implementation and to support personnel training. Beyond the immediate costs, lack of standardisation presents another issue. As will be demonstrated in the following sections, although international standards do exist, their implementation can vary depending on the tools adopted, leading to inconsistencies.

Finally, integrating MBSE with traditional methodologies can be quite difficult, especially when it comes to converting existing documentation into detailed models. The following table 1.2 provides a summary of the advantages and disadvantages of this methodology.

**Table 1.2:** Strenghts and weaknesses of MBSE

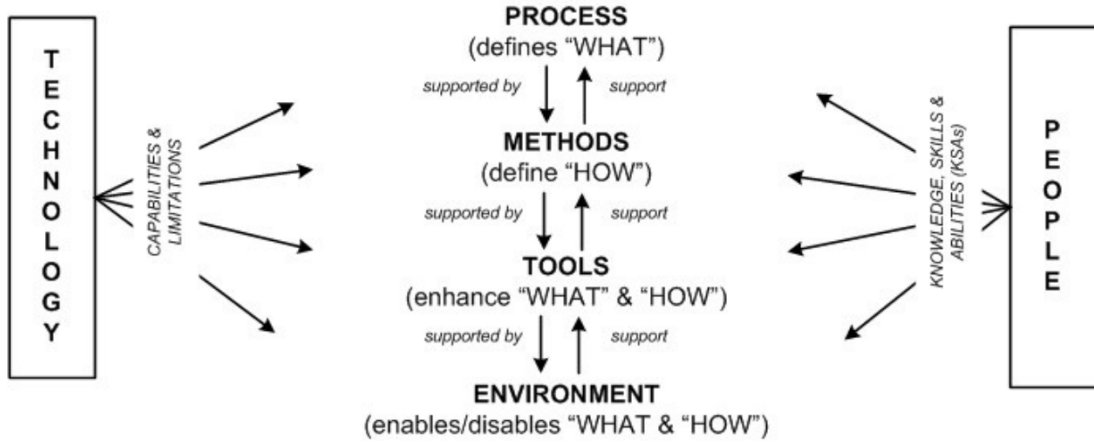
MBSE features	
Strenghts	Weaknesses
Increased productivity	Steep learning curve
Enhanced communication	Initial high costs
Improved quality	Lack of standardisation
Complexity management	Complexity in system migration
Early defect detection	

The following sections will delve more deeply into the principles of MBSE, a methodology that, in comparison with DDSE, has been extensively described in the literature. Firstly, the MBSE methodologies currently employed in the industrial sector will be presented, followed by a subsequent section dedicated to the languages employed.

### 1.3.1 Methodologies

According to J. A. Estefan in [7], the word *methodology* can be defined as a collection of related processes, methods and tools. A *process* is a logical sequence of tasks designed to achieve a specific objective. It outlines "WHAT" needs to be done, without clarifying "HOW" each task is executed. Processes are structured in layers, allowing for analysis and definition at various levels. A *method* provides the techniques for performing a task; it defines the "HOW" for each task. Every task within a process is executed using methods. A *tool* is an instrument that enhances the efficiency of a task when applied correctly to a specific method. Its primary purpose is to make performing the "HOWs" easier. In systems engineering, most tools are software-based, allowing creating models using appropriate languages. Tools should also integrate with each other to ensure completeness, correctness and consistency across the project.

There is also the *environment*, defined as the surrounding circumstances, external objects, conditions, or factors that impact the actions of an object or an entity. These conditions may be categorised as social, cultural, personal, physical, organisational, or functional. In the context of a project, the environment is designed to facilitate the effective integration and support of its tools and methods. [7]



**Figure 1.2:** The PMTE Elements and Effects of Technology and People. [3]

As can be seen in the figure 1.2, when selecting the right mix of Processes, Methods, Tools and Environment (PMTE) elements, both the capabilities and limitations of the technology and the knowledge, skills and abilities of the people are included.

As already mentioned in the previous section, MBSE is about elevating models in the engineering process to a central and governing role in the specification, design, integration, validation, and operation of a system. A variety of models exist to represent the different aspects and types of a system: from models that, in the early life cycle of a system, define different mission and system concepts, to models that support system design solutions. But also models that support the integration of software and hardware components, together with system verification, can be implemented. [7]

Several approaches have been developed in recent years to support MBSE and to implement it. A summary of the leading MBSE methodologies, as detailed in the Handbook [4], is presented below.

### **INCOSE Object-Oriented Systems Engineering Methodology (OOSEM)**

It has been developed by the INCOSE since 1998. It can be defined as a "scenario-driven approach" [4], integrating the process in a fully object-oriented context within the model-based approach. A fundamental aspect of the methodology is the delineation of the system's objectives, mission, and operational scenarios. The V-diagram is the most suitable model for the implementation of the product life cycle. It adopts the Systems Modeling Language (SysML) language for the specification, analysis, design and verification of the system.



**The IBM Rational Telelogic Harmony-SE** The process is based on the “V” life cycle model, using an iterative workflow through the three phases of requirements analysis, system functional analysis, and design synthesis. It supports the Model-Driven Development (MDD) approach, where the model is the central work product, based on the SysML structure diagrams. However, IBM provides support for Harmony within the IBM Rational tools packages.

**The IBM Rational Unified Process for System Engineering (RUP-SE)** It enhances the Rational Unified Process Rational Unified Process (RUP) of concurrent design and iterative development for Model-Driven System (MDS) through the utilisation of the spiral model and object-oriented principles. The approach employs business modelling, the first discipline of iterative tasks, to ensure the system’s conformity to business activities. The product is characterised by the presence of either a series or a single product with multiple versions. It is also defined by market permanence and service.

**The Vitech Model-Based System Engineering** This approach involves Vitech’s CORE® environment as a system design repository, connecting stakeholders, source requirements, behaviour and architecture domains, and verification and validation. The approach is founded on a model of product development, the Onion Model, composed of layers that enable users to progress from the source documents of the customer to the final specifications. The language employed for technical communication is the System Definition Language (SDL), which is predicated on relationships, entities and attributes.

**The Jet Propulsion Laboratory (JPL) State Analysis (SA)** The JPL has established a methodology for defining model- and state-based control architecture. The states delineate the conditions that the system attains as it evolves over time, while the models specify the evolution of the system states themselves. This method supports the evolution of the system model during the project life cycle, through an iterative modelling process and the state analysis information, which are collected in a Structured Query Language (SQL) database.

**The Object-Process Methodology (OPM)** It is a "holistic system paradigm" [4] based on *objects*, defined as entities which simply exist, and *processes*, which are described as transformation patterns applied to the objects. The methodology delineates the system development, life cycle support, and evolution through three stages: requirement specification, analysis and development, and implementation. The OPM integrates the Object-Process Language (OPL) (process-oriented approach) with the Object-Process Diagram (OPD) graphic model (object-oriented approach), supported by the OPCAT software environment.

**The ARCADIA Method** It is a model-based engineering methodology for systems, hardware and software architecture design, which has been created by Thales between 2005 and 2010. It is recommended that three mandatory interrelated activities are implemented: Need Analysis and Modelling, Architecture Building and Validation, and Requirements Engineering. The ARCADIA is supported by a standard modelling tool (the Melody Advance/Capella) that depends on the Unified Modeling Language (UML) and SysML languages.

**The Systems Modeling Process (SYSMOD)** It is a user-oriented methodology for requirements engineering and system architectures. It involves stakeholders analysis, requirements and system context definition, requirements analysis, domain model definition, and system architecture definition by levels (functional, logical, physical). It relies on the SysML modelling language.

### 1.3.2 Modelling languages

In the SE context, while numerical modeling can rely on a solid mathematical apparatus, functional modeling has long suffered from a lack of suitable tools. Fortunately, the development of meta-models based on intuitive graphical languages has filled this gap, allowing designers to communicate effectively with users through the functional model. This summary will explore the main languages used in SE, with a particular focus on UML and SysML.

In the domain of software engineering, the design by objects was effectively supported by the UML language, which was subsequently elaborated, enhanced and adapted to the system design as the SysML. UML's primary objective was to create a language that did not rely on mathematics, but on standard diagrams and modeling elements with defined rules and semantics. It allows for a clear description of activities, architecture, operational rules, and interactions (even with the user) through a graphical meta-model.

The adaptation of UML to more general and complex systems led to the birth of SysML. Developed starting in 2001, with significant contributions from INCOSE, SysML is a true customization of UML for systems engineering. It distinguishes itself from UML by introducing modeling *blocks* instead of classes and robustly supports the modeling of requirements, structure, behavior, and parameters of a system, its components, and its environment. It's fundamental for the MBSE, as it permits key concepts to be visualized through simple effective diagrams. Each *block* serves as the basic structural unit of the model, representing a system element. [4]

SysML is structured into four main types of diagrams: (i) *requirements diagrams*,

specifically introduced in SysML to link the system model to the preliminary elicitation of requirements, a crucial aspect of systems engineering; (ii) *structure diagrams*, which describe the composition of the system, its components, and their relationships; (iii) *behavior diagrams*, which illustrate dynamic interactions, activities, and flows within the system; (iv) *parametric diagrams* that represent a significant innovation, enabling the quantitative modeling of system behavior and architecture, essential for numerical simulations and optimization.

More recently, new updates and evolutions of those languages are proposed to more properly fit some specific applications and domains. A lack of entities in the original UML and SysML has been identified, particularly in sectors like industrial engineering. This has led to the research and development of new generations of communication and modeling tools, such as the Interdisciplinary Modeling Language (IML) or the Automation Modeling Language (AutomationML). Their main goal is not specifically the SE, but their focus is predominantly on the technical domains in which they are currently engaged. In summary, while SysML remains a prevalent tool for SE due to its robustness and versatility, the emergence of new languages and ongoing research indicates a field in constant evolution, requiring increasingly specific and powerful tools to address the growing complexity of modern systems. [4]

## Chapter 2

# Systems Engineering Tools

The present chapter is dedicated to the analysis of tools suitable for implementing SE, which are a fundamental part of the present thesis project. As mentioned in the previous chapter, the implementation of SE necessitates the utilisation of software tools for the collection of requirements and the realisation of the modelling activity delineated in [4].

Firstly, an introductory overview of the most significant SE tools currently available on the market is presented. Subsequently, the focus is directed towards a detailed exposition of two environments selected for the purpose of integration.

The following list outlines some of the most relevant software programs.

**IBM Rational DOORS - Rhapsody** The IBM Corporation has designed IBM Rhapsody to work together with DOORS to track and manage design requirements throughout the project's life cycle and to enable navigation between design and requirements. DOORS is currently used for the requirements elicitation, while Rhapsody implements the typical tools of the SysML language, performing system specification and modeling. The interface works by sharing information between the Rhapsody model and the DOORS database. [4][12]

**Cameo System Modeler** Dassault Systemes has developed an industrial MBSE environment which provides tools for the creation of SysML models and diagrams. The software supports the analysis of the design, the requirements verification, and the decision-making process, while preserving model consistency and monitoring design progress. Its key benefits are: requirements management, traceability among different levels of the project, reports customisation, resolution of parametric models, development and configuration management. [13]

**System Composer** It is an MBSE tool, developed by MathWorks, for the purpose of defining, analysing, and simulating system and software architectures. It deeply integrates with the MathWorks ecosystem, including Simulink for detailed behavioral modeling, simulation, and design validation using various diagrams and models, Embedded Coder for direct code generation from software, and Requirements Toolbox for requirement allocation and traceability. It offers robust analysis capabilities for trade studies and comprehensive documentation generation. [14]

**Papyrus** It is an industrial-grade open source MBSE tool that offers support for both SysML and UML languages. It has notably been used in industrial projects and is the base platform for several industrial modeling tools. It enables engineers to create complex models of systems and to support the entire modelling life cycle, from requirements specification and architecture definition to analysis and simulation via additional plugins, ensuring data traceability during the project. [15]

**Valispace** The DDSE web-based platform is utilised by engineers for the creation of logical models of a product, in addition to the management of requirements, technical parameters, and design modifications. Technical properties are stored in a single database, with formulas connecting them. It allows engineers from different disciplines and external stakeholders to work simultaneously on a project. Moreover, the single database provides consistency throughout all project phases. It easily integrates with other engineering tools through an API, thus facilitating the tracking of relationships that are situated outside the tool itself. [16]

**Capella** It is a comprehensive MBSE tool and method to specify and design system architecture, and to manage complexity. It is based on the ARCADIA method. It provides a conceptual framework for the analysis and modelling of system architecture at different levels of abstraction, from operational and system analysis to logical and physical architecture. It offers diagrams and viewpoints to support the engineering process, facilitating collaboration, consistency and traceability. [17]

This was a brief summary of the SE tools that have been most frequently cited in the literature and that appear to be the most widely used in industry.

The next sections focus on the two main tools upon which the thesis is based: Valispace, for DDSE implementation, and Capella, for MBSE. A more detailed description of the features and capabilities of the two software instruments is provided, as well as the method they implement.

## 2.1 Valispace

DDSE focuses on managing all engineering information as interconnected data points within a single source of truth, and Valispace is a web-based platform that perfectly embodies and enables the DDSE philosophy. It acts as a central hub for all engineering data, moving beyond outdated documents to manage dynamic information. [6].

This approach empowers engineers with a complete, real-time overview of project data and status. It enables the development of logical models, with all their technical properties being stored in a unified, consistent database, with the relevant formulas connecting these properties. This fundamental feature significantly differentiates Valispace from current solutions. The integration of properties with formulas facilitates the immediate visualisation of parameter dependencies, thereby providing a comprehensive understanding of the impact of design modifications on the system. Furthermore, this tool has been demonstrated to significantly enhance collaboration, aiding all engineers engaged in a project in working with consistent project data. It facilitates the exchange of data with suppliers and customers, providing a comprehensive overview of complex systems in real-time and ultimately reducing overall development time. Moreover, Valispace supports the reuse of design information between projects, further increasing efficiency. [16]

An thorough investigation into its main strengths reveals that Valispace is, above all, characterised by its **ease of use**. The tool does not require training for its use, due to the intuitive web interface design. It has been designed to incorporate a user-friendly workflow, with the objective of reducing the time and effort required for training, thus mitigating one of the disadvantages of current SE tools. As the model becomes more intricate, users can gradually familiarise themselves with the tool's functionality.

Another feature that can be observed is the **consistency**. Valispace is based on a consistent database. The identification of discrepancies during the early stages of a project is facilitated, thereby avoiding costly and time-consuming consequences that may require additional expenses and rework. Furthermore, the user is automatically informed of any alteration of the engineering data via an integrated notification system.

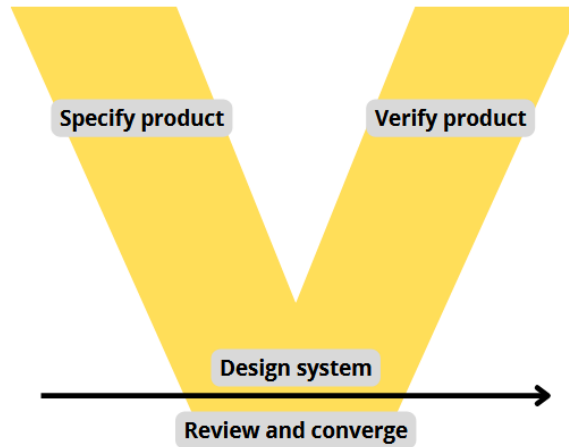
As a browser-based tool, it does not necessitate installation on a specific platform. It can be deduced that **flexibility** represents a further strength. Systems engineers have the potential to access data remotely via web browser and to create, store and manage engineering data from anywhere. Additionally, the exportation, importation and direct access of data from alternative tools is facilitated by a dedicated API that enables automated processes and scripting capability.

Finally, the adaptive nature of the web interface allows the configuration of diverse

perspectives that are aligned with the requirements of different stakeholders. It is not necessary to comprehend a specific MBSE methodology or modelling language in order to obtain pertinent information about the system. [16]

Valispace optimises the engineering process from requirement definition to system design, testing, verification and validation. This systematic methodology is guided by the principles of the "*V model*", as shown in figure 2.1. [18]

Starting from the **Product specification**, Valispace allows the integration of



**Figure 2.1:** Valispace V-model [18].

engineering data ( e.g. mass, volume and power consumption) with formulas, calculations and simulations through the parametrization of requirements. Indeed, it makes the requirements dynamic, associating them with technical properties, thus guaranteeing that calculations are automatically updated in the event of any change.

In the context of **System Design**, Valispace supports the interaction of physical properties and requirements. Consequently, any changes made will be visible to all members of the engineering team, together with the effect that component variations have on the system as a whole. The tool supports not only rapid design iterations, but also trade-offs. Furthermore, by connecting requirements and components to formulas and calculations, it guarantees that the team is always dealing with the latest data, eliminating the need to check numerous other documents to verify the correctness of the data.

In Valispace, the "*V model*" is characterised by the automatic and real-time execution of **Product Verification**. This process occurs in conjunction with the progression of the system model, and by establishing interconnections between requirements, physical units, parameters and calculations. Concurrently, the

collection, analysis, and integration of requirements and performance data take place.

The system is capable of concurrent management of two distinct data types: requirements data and performance data. These two types of data are continuously checked against each other through automatic verification. Moreover, Valispace allows the creation of customised test procedures based on specific criteria. Once the execution of the requirements has been completed, their verification will be conducted automatically.

Finally, it is possible to conduct **Reviews** of the data. It is important that all the teams have the capacity to execute and verify any actions that come from reviews conducted directly on the data rather than in additional documents. The efficacy of this system in reducing the time engineers spend retrieving documents has been demonstrated, thereby ensuring that all team members are aware of the actions they need to take.

The Valispace platform is articulated in the following sections, as shown in figure 2.2.



**Figure 2.2:** Valispace overview.

**Project Module** It provides a project management overview. Users can have access to a range of features to manage the project timeline and to coordinate with colleagues.

**Components Module** It shows the mission architecture and the product tree of the components constituting the system. The tool lets the user assign different types of properties to each component. These are called *valis* for numerical values and formulas, *dataset* for varying values, *textvali* for text, *datevali* for date and *matrix* for properties with different values per state. It is also possible to associate *modelists* with components, which represent the operational modes of the component in question and which, together with the other properties, will then allow project budgets to be calculated. Another option provided by Valispace is the ability to create trade-offs, offering a clear view of the different alternatives with their respective parameters, thus enabling the most suitable option to be chosen.

**Requirements Module** It provides an intuitive and effective way to manage the project's requirements. They can be assigned to components or components'



values, and their verification status can be tracked throughout the design phases. A requirements hierarchy can also be outlined and visualised clearly due to the parent-child relationships.

**Analyses Module** It gives users the ability to write and manage all the project documents, including graphs, tables, budgets and reports that will be stored in this module.

**Scripting Module** It allows users to write and run simulations using the Octave or Python programming languages. More complex calculations are what the module is designed to perform.

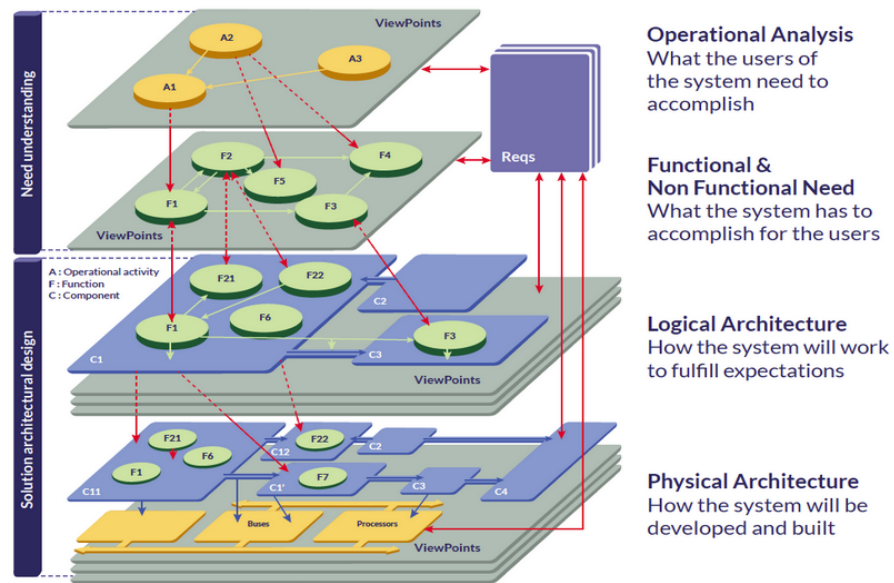
**Tests Module** It provides a single place to perform all verification activities, from creation to organisation and tracking. Users can create procedures to verify requirements, equipped with attached files and tags.

## 2.2 Capella

Capella represents an open-source solution for MBSE, that adopts the ARCADIA method. The ARCADIA methodology was developed by the Thales Group and has been employed since 2011 in several projects across a variety of domains, including satellite systems. This "*structured engineering method*" is dedicated to the definition and validation of the architectures of complex systems [19]. It promotes collaboration between all stakeholders, encouraging an enhanced understanding of real customer needs and facilitating the sharing of product architecture. It enables the execution of iterations from the definition phase onwards, thereby ensuring the alignment of the architecture with all identified needs.

The ARCADIA method is founded on a series of "*Golden Rules*" [17] that have been established to ensure the robustness and feasibility of architectural design. The integration of need analysis, requirements and architecture definition and validation as essential and interrelated activities is central to the approach.

This method is here elucidated with particular regard to its fundamental principles. The figure 2.3 provides a visual representation of its main engineering levels.



**Figure 2.3:** The main engineering levels of the ARCADIA Method [17].

The following is an overview of these levels [19].

**Operational Analysis** "What the users of the system need to accomplish." [19]  
This encompasses more than simply meeting system requirements; it concerns a detailed understanding of the problem from the user and stakeholders' perspectives, before any thought of a technical solution. The process involves the identification of the needs, the establishment of the relevant actors and their respective roles, the formulation of high-level objectives to be accomplished, and the definition of the specific activities to be performed by the actors in order to achieve these aims. The main output is a clear understanding of the overall stakeholders' needs, which is achieved by creating a clear overview of the mission as it is envisioned, independent of any system that might eventually support it.

**System Analysis** "What the system has to accomplish for the users." [19]  
In this sections, the focus shifts from the needs of the user to the needs of the system itself. The objective is to define how the system will satisfy those needs and what its expected behaviours and qualities will be. At this stage, system functions and data exchanges are delineated, but also non-functional constraints and interactions between systems and operators are clarified. The primary purpose is to verify the feasibility of customer requirements. The main outputs include system functions, interactions between users and external systems, and consolidated

system requirements. This phase is crucial for specifying the design and requires customer validation.

**Logical Architecture** “*How the system will work to fulfill expectations.*” [19]

This entails an internal functional analysis, which involves the decomposition of system functions into sub-functions and the identification of logical components. The initial step in this process is to define the solution’s expected behaviour using functions, interfaces and data flows, taking into account both the system’s functional and non-functional needs. These functions are then allocated to one or more logical components. The result is a detailed, structured logical architecture that comprises components, justified interfaces, scenarios, modes and states.

**Physical Architecture** “*How the system will be developed and built.*” [19]

This level shares the same core objective as the Logical Architecture, that is to define how the system will fulfill its expectations. However, it establishes the final concrete architecture of the system. This perspective introduces the necessary details and design decisions for implementation. It incorporates all the functions required by the specific technical choices and implementation details, while also highlighting behavioral components (e.g. software components) that will perform these functions. The primary output of this engineering phase is the selected physical architecture, detailing the components that need to be produced.

**End Product Breakdown Structure (EPBS)** “*What is expected from the provider of each component.*” [19]

It represents the final and most detailed level of architectural definition. In this section, the conditions that each component of Physical Architecture must fulfil in order to satisfy the constraints and design choices are identified. This level breaks down the physical architecture, thereby ensuring that each individual component meets its specific performance requirements and limitations. This, in turn, facilitates its integration into the system design. This phase is crucial for preparing a robust and secure Integration, Verification, Validation, and Qualification (IVVQ) process.

The method can be both top-down or bottom-up, depending on the project’s status. The ARCADIA method supports iterative and recursive application across the system’s hierarchy. Any subsystem can be considered the "system" for the next level of analysis, a breakdown that continues until specific subsystems or Commercial-Off-The-Shelf (COTS) are reached. [19]

ARCADIA’s objective is to transform the engineering field through the implementation of a model-based approach, that diverges from the conventional

document-centric methodologies. In order to achieve this result, Thales developed a unified architectural modelling language, Capella, starting in 2007. In contrast to most of the companies that adopted existing tools, Thales prioritized developing the ARCADIA method, based on functional analysis and the allocation of functions to architecture elements, something more familiar to engineers. As a result, Capella intuitively integrates both the method and its associated language. This synergy enables engineers to focus on defining architectures rather than dealing with complex modeling languages like UML or SysML. Capella is a comprehensive framework encompassing the full range of engineering activities, from the initial identification of client requirements to integration, verification, and validation. [19]

A detailed analysis of Capella's characteristics reveals that one of its key strengths is its ability to facilitate **enterprise-wide collaboration** and **co-engineering initiatives**. All engineering teams share a common methodology, a unified exchange of information, and a model. This shared model is fundamental to the formalisation of the analysis of operational and functional needs, as well as the definition and justification of the architecture models. The Capella models, constructed for each ARCADIA engineering phase, are integrated through transformations and connected by justification links, enabling comprehensive impact analysis, particularly during evolutions.

Moreover, Capella has been designed for the purpose of architectural design, relying on a **Domain-Specific Language (DSL)** rather than general languages like UML or SysML, making it easier for users. It manages and scales complexity through its abstraction levels, facilitating model maintenance, large-scale modelling and enabling model evolution and reuse.

Another fundamental aspect of the proposed framework is the focus on the **early verification of architectural design** to ensure the robustness and feasibility of the design itself. The method establishes a set of rules for preliminary verification, with the aim of validating architectural design in a way that is efficient in terms of time and cost.

The final advantage delineated is the ability to adapt to different projects and organizational needs, ensuring that this solution is not a rigid solution but a **flexible framework**. Capella balances design drivers (like functionality, performance, cost, safety and security) and constraints to optimise designs and handle different product configurations. This is what renders Capella suitable for this approach, in terms of adaptability and flexibility. [19] [17]

The following areas are shown when entering the Capella interface [19].

**Activity Browser** It provides guidance through the various engineering phases of your architectural modeling. The software facilitates the generation of new diagrams, while providing a means of creating "transitions" between different

phases.

**Semantic Browser** It assists in navigating the model. When an item is selected, the Semantic Browser displays all its references, including contained and related elements, and every diagram related to it.

**Project Explorer** It is a traditional tree-view diagram that shows the entire Capella model. It provides an overview of all the items and diagrams.

**Diagram** It presents a graphical representation of a specific component of the model, allowing the processes of creation, modification, and deletion of items, as well as adjustment of their organization or aesthetic characteristics.

**Properties** This section displays all the properties associated with any selected item.

In conclusion, a brief summary of the types of diagrams available in Capella is presented in the table 2.1.

**Table 2.1:** Overview of types of diagrams available in Capella [17].

Diagram Type	Description
<p><b>Capabilities</b></p>	<p>Available in all engineering levels. Used to manage relationships among <i>Missions</i>, <i>Actors</i>, and <i>Capabilities</i>.</p>

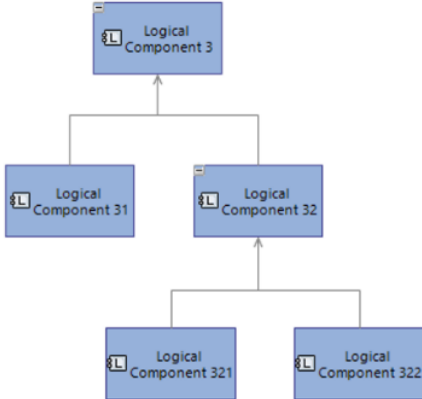
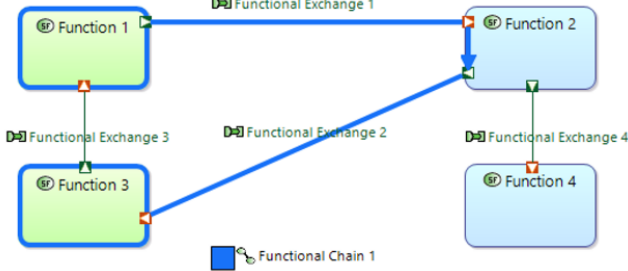
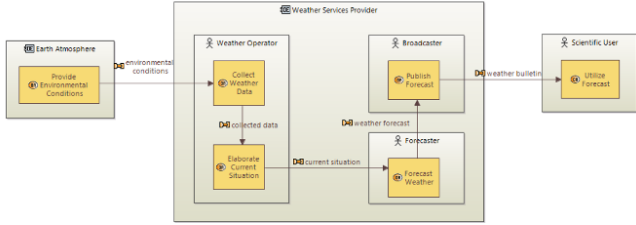
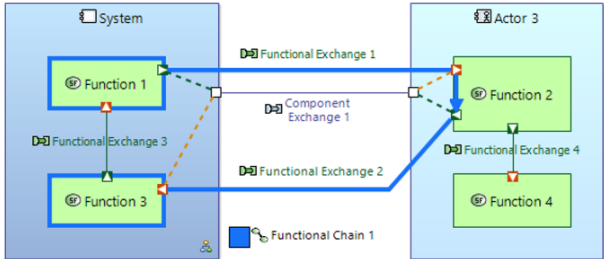
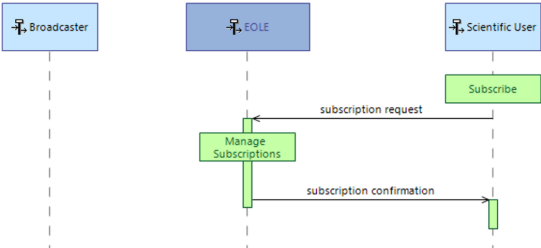
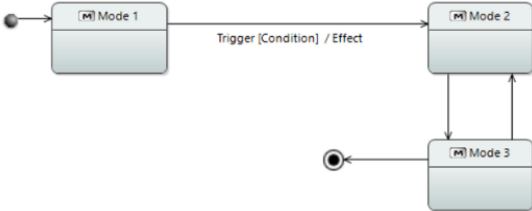
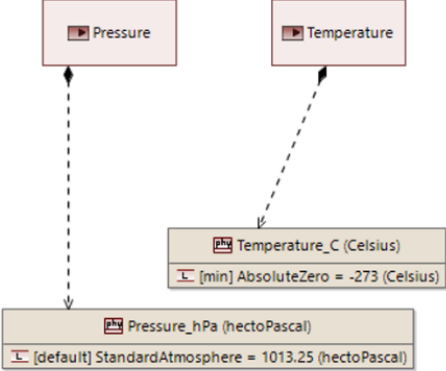
Diagram Type	Description
<p style="text-align: center;"><b>Breakdown</b></p> 	<p>Present at all levels. Show hierarchical decomposition of <i>Functions</i> and <i>Components</i>.</p>
<p style="text-align: center;"><b>Data Flow</b></p> 	<p>Represent exchange of information between functions. Enable highlighting of <i>Functional Chains</i>. Used across all levels.</p>
<p style="text-align: center;"><b>Operational Architecture</b></p> 	<p>Built at the Operational Analysis level. Allocate <i>Operational Activities</i> to <i>Entities</i>. Highlight <i>Operational Processes</i>.</p>
<p style="text-align: center;"><b>Architecture</b></p> 	<p>Used in all levels. Allocate <i>Functions</i> to <i>Components</i>. Highlight <i>Functional Chains</i>.</p>

Diagram Type	Description
<p style="text-align: center;"><b>Scenario</b></p> 	<p>Represent "vertical sequence of messages" exchanged among elements. Multiple variants available.</p>
<p style="text-align: center;"><b>Mode and State</b></p> 	<p>Represent UML/SysML-like state machines. Link to <i>Functions</i>, <i>Exchanges</i>, and other concepts.</p>
<p style="text-align: center;"><b>Class</b></p> 	<p>Model precise data structures. Associate with <i>Functional Exchanges</i>, <i>Components</i>, and more.</p>

## Chapter 3

# Valispace-Capella Integration

This chapter outlines the methodology employed by this thesis, which is based on the complementary nature of the DDSE and MBSE approaches, as well as the integration of Valispace and Capella tools.

After an initial section that provides a justification for the choices made regarding the adopted methodologies, the core of the integration process is explained. The steps and the roles of the two tools are then highlighted, along with the modalities of their integration.

Finally, the scripts created to export engineered data from Capella for import into Valispace and to compare and ensure conformity between the two environments are detailed.

### 3.1 DDSE-MBSE approach

MBSE has become a widely adopted methodology for the design of complex systems, offering substantial improvements in terms of efficiency and efficacy compared with traditional document-based approaches. However, MBSE tools are currently considered excessively complex as they attempt to address the numerous technical challenges faced by systems engineers. To overcome these limitations, the emerging approach of DDSE has been introduced to complement MBSE. DDSE simplifies the design process and improves collaboration. In particular, it works with engineering data and its associated structures and links to form the basis of the systems engineering process.



The present thesis aims to develop a methodology that is more agile and flexible than traditional approaches. This research project is inspired by the concept outlined in the paper [20] by Elecnor Deimos, which aims to develop a methodology that exploits the complementarity of DDSE and MBSE, and consequently of multiple tools, with the objective of reducing development costs and times without the necessity for extensive training of personnel. Deimos asserts that this methodology has been successfully employed in numerous missions, encompassing both Earth observation, deep space and scientific missions [20].

The integration of these approaches produces a powerful combination, capable of compensating for each other's weaknesses. In consideration of the conclusions reported in tables 1.1 and 1.2, which provide a detailed exposition of the respective benefits and limitations of the individual methodologies, it is possible to proceed with the critical analysis of the complementarity of the two methodologies, that is summarised in table 3.1.

**Table 3.1:** DDSE and MBSE integration.

	<b>DDSE</b>	<b>MBSE</b>	<b>Integration</b>
<b>System representation</b>	Engineering data	Models	Models provide context for data, that validates models
<b>Complexity handling</b>	Management of datasets and requirements	Hierarchical decomposition of components and functions	Management of complexity from different perspectives
<b>Decision making</b>	Decisions based on simulations and analysis	Decisions based on system design and architecture	Decisions based on analysis results and design choices
<b>Traceability</b>	Traceability of data, results and requirements	Traceability of models	Enhanced traceability of the project through integration
<b>Ease of use</b>	Easy to learn, user-friendly procedure	Steeper learning curve	Initial complexity of MBSE mitigated by familiarity with DDSE

The MBSE provides the architecture and functional connections, thus demonstrating the system's functionality. The model created in Capella serves as a reference structure for the DDSE in managing the data and the requirements derived from it. Conversely, the DDSE supports MBSE models through the definition of concrete engineering data that can be used for analyses and simulations, thereby enhancing the completeness of the project. Together, these tools provide comprehensive support throughout the entire mission life cycle of the product. Moreover, their integration allows for more comprehensive management of project complexity. The

DDSE provides consistent management of all collected data, while the MBSE breaks down the various components and their functions, making the models easier to understand.

The decision-making process and traceability are also improved, with the two methodologies complementing each other in terms of analysis and simulation results and design choices. Finally, the ease with which the DDSE can be adopted, given its widespread use among companies, mitigates the initial difficulty of approaching the MBSE methodology.

Their combination ensures the continuity of information and the preservation of a "single source of truth." The objective of the integration is to illustrate the efficacy of the design process by allowing for rapid change management, thereby increasing both effectiveness and efficiency.

In view of the numerous challenges and tasks with which the field of SE is confronted, the adoption of a single tool often proves insufficient [20][21]. The present thesis implements this approach using two software tools, Capella for MBSE and Valispace for DDSE, thereby pursuing continuity and conformity between the two through continuous data exchange throughout the entire project. The two tools selected for implementation, conversely, aspire to attain a more robust design, leveraging each other's strengths and limitations. While Capella demonstrates proficiency in architecture definition, functional analysis and operative modes definition, Valispace offers a user-friendly environment, capable of managing engineering data and requirements, in addition to budget estimation, in real time.

The following table 3.2 summarises the main strengths and weaknesses of the two tools.

**Table 3.2:** Valispace and Capella capabilities [20].

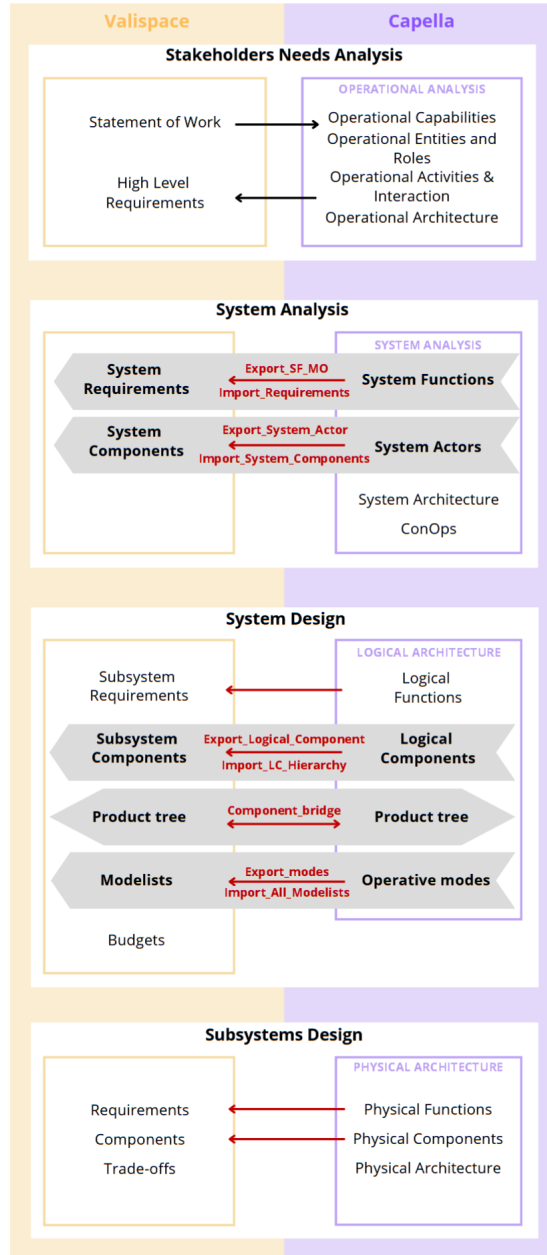
	<b>Valispace</b>	<b>Capella</b>
<b>Requirements</b>	Excellent for requirements management, including digitization, flow-down, traceability, and ease of export; highly user-friendly.	Strong in requirements management, including propagation and traceability with model elements.
<b>Concept of Operations</b>	Poorly suited.	Very strong for defining ConOps through sequence diagram and VPMS Add-On.
<b>Modes and States</b>	Not very strong, but it allows for modes definition and allocation to components and their values (useful for budgets).	Strong for detailed modeling of modes and states with triggers and transitions.

	Valispace	Capella
<b>Functional Analysis</b>	Poorly suited.	Very strong for modeling of functionalities and functional interfaces at various levels through the use of breakdown diagrams.
<b>Architectures</b>	Weak, it allows the physical architecture definition but it is intended for use in engineering data management. Useful for trade-offs.	Very strong for logical and physical architecture definition, functions allocation to components and functional chains.
<b>Budgets and Simulations</b>	Very strong for modeling components with "non-static sizing values", providing budgets and simulations.	Weak, its main focus is on static data related to components.
<b>Team collaboration</b>	Very strong for real-time collaboration, including reviews, notes, and notifications.	Strong with the support of Add-Ons.
<b>Real-time coordination</b>	Very strong, it supports real-time coordination, with constantly updated information shared among all participants.	Weak, it allows shared models, but not in real time.

## 3.2 Integration

In view of the considerations expressed in the previous section about the DDSE-MBSE approach, this section elaborates on the integration process of the two tools, Valispace and Capella, the responsibilities of each of them, and the measures implemented to guarantee compliance between the two. The figure 3.1 provides a high-level overview of the process, which will now be analysed in detail.

Firstly, it is essential to clarify the roles of the two tools: Capella will be established as the master, the principal entity responsible for the mission and system architecture, while Valispace will fulfil a supporting role, managing all engineering data, requirements and budgets. It is important to note that Valispace will be the entity that must adapt to the changes and choices made by Capella. In order to achieve this objective, a number of Python scripts have been developed. These scripts execute two main functions: firstly, they enable the export of data from Capella, and secondly, they allow the subsequent import of this data into Valispace. Furthermore, these scripts verify the consistency between the two at the system level. To implement this solution, it was determined that the Capella add-on, designated *Python for Capella* [22], and the Valispace API [23] would be utilised. These will be explored in greater depth in the following sections.



**Figure 3.1:** Tools integration process.

The next paragraphs will analyse the integration process step-by-step. As shown in figure 3.1, the two environments Valispace and Capella have been divided to highlight the functionality attributed to each. In addition, the various engineering

levels are presented from the top to the bottom according to the ARCADIA method.

It is possible to discern different types of arrows. The first type of red arrow displays a script above it, which defines the implementation of the Python code and, consequently, the establishment of the connection between the two environments. The red ones without any reference above represent a possible connection between the two environments via codes not implemented in this study. Finally, the black arrow simply defines the flow of the work.

The following discussion will proceed through a gradual examination of each respective level.

### Stakeholders Needs Analysis

The initial phase of the project begins with the Statement of Work (SoW), which delineates the objectives and constraints of the project. The initial mission and customer needs are then identified and defined as requirements on Valispace. The subsequent phase involves conducting the actual **Operational Analysis** on Capella, encompassing the stakeholder needs analysis, the identification of the entities involved, and the allocation of related activities to them, thus producing the *Operational Architecture*. The diagrams generated on Capella will facilitate the definition and refinement of the high-level requirements on Valispace, also providing the necessary context for their consolidation during the system's architectural definition on Capella. In this particular instance, the implementation of a script was not considered a required measure to ensure the consistency of the two environments. It is evident that the stakeholder needs analysis is more successfully executed on Capella. Conversely, Valispace exclusively handles the management of the requirements themselves.

### System Analysis

The objective of this level is to establish the configuration of the system that will satisfy the requirements of the users. In this context, the process begins with the **System Analysis** on Capella, where a functional analysis is conducted and the involved actors are determined. In this case, two sets of scripts are implemented to facilitate the process.

Starting from the functional analysis, the **Export\_SF\_MO.py** script (3.3) exports the functions of interest modelled on Capella and the **Import\_Requirements.py** (3.4) imports them into Valispace in the form of *Requirements*. Among the other scripts, **Export\_System\_Actor.py** (3.3) exports the actors defined on Capella and **Import\_System\_Components.py** script 3.4 imports them into Valispace as *Components*, thereby enabling the

definition of a compliant mission architecture in the DDSE environment.

The next step of the process is the elaboration of the detailed *System Architecture* on Capella, along with the definition of the Concept of Operations (ConOps) in the same environment, through the use of the ViewPoint Modes and States (VPMS) Add-On [24]. As previously observed, the diagrams developed in Capella help to consolidate and satisfy the requirements stored in Valispace. This statement is applicable to all levels.

## System Design

The process at this level also begins with Capella's **Logical Architecture**, which defines the detailed internal logical structure of the system, enabling the decomposition of the functions and their allocation to *Logical Components* (in this case, the subsystems of the system). In this phase, there are multiple scripts available to facilitate the integration. As with the previous level, the first two sets of Python scripts follow the same logic as the two implemented in the System Analysis.

Although not implemented in this particular instance, it is possible to establish new requirements on Valispace, starting from the *Logical Functions* defined on Capella, by making minor changes to the scripts **Export\_SF\_MO.py** and **Import\_Requirements.py**.

The second set of scripts is responsible for the exportation of the complete hierarchy of *Logical Components* created on Capella through the **Export\_Logical\_Component.py** script 3.3, subsequently bringing them into Valispace using the **Import\_LC\_Hierarchy.py** code 3.4, while ensuring the maintenance of the hierarchy.

At this point, a "*bridge script*" has been formulated in order to ensure conformity between the two environments. The script in question is **Components\_bridge.py** 3.4 and its function is to compare the entire hierarchy of components (i.e. the *Product Tree*) established in the two tools, compare them, and show a report to the user underlining the discrepancies between the two environments. Consequently, if a component appears to be missing in Valispace when Capella is considered the master, the missing component will be imported into Valispace automatically to restore conformity.

Another activity undertaken on Capella has been the definition of the modes and states diagram, which facilitates a detailed analysis of the operative modes. These modes are exported from Capella through the **Export\_modes.py** script 3.3, and then imported via the designated script **Import\_All\_Modelists**

3.4 on Valispace as *Modelists*, and assigned to the component of interest.

The final steps involve the development of a *Logical Architecture* on Capella, with the allocation of functions to components, and the estimation of budgets (mass, volume, power) on Valispace. The latter process exploits the imported operative modes and the engineering data, already defined and assigned to the components on Valispace.

### Subsystems Design

The final level to be implemented is the physical one, which, in this case, has not been completely developed but has barely been started with the aim of showing the process and of completing it in the future. The objective of this phase is to define the final physical architecture of the system, including the selected physical components and their technical characteristics.

The procedure starts on Capella with the definition of the **Physical Architecture**. In this particular case, the development of a single subsystem has been conducted with the purpose of demonstrating certain functionalities of the process. As in the previous phases, the requirements defined on Valispace are consolidated by the architectures defined on Capella.

The same scripts that facilitate the establishment of requirements and the transfer of components from Capella to Valispace are also applicable at this level, with minor changes.

This level also allows for an increasingly detailed analysis of the system, both of the architectural model and of the operative modes, which can now be applied to the physical components, consequently creating more detailed budgets on Valispace.

Furthermore, it is important to demonstrate Valispace's capacity to efficiently manage trade-offs between components.

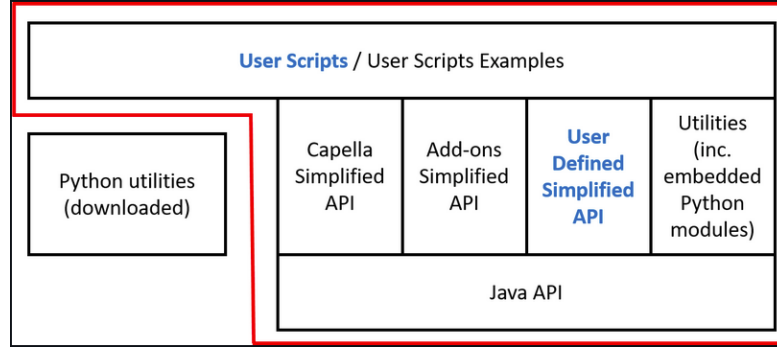
## 3.3 *Python for Capella*

*Python for Capella* is an open-source add-on of Capella that enables interaction with models built on Capella via Python. This solution supports the creation of Python scripts, which are utilised for the extraction and importation of information from and about the model itself. Given that the extraction of model information constitutes a crucial step in the execution of the project, the add-on proved to be very useful.

Technically, *Python for Capella* includes Eclipse EASE to allow script execution, PyDev for script edition and finally a Python Interpreter. It therefore supports the use of a common coding language, Python, and provides high-level example scripts that are accessible to the user but can be totally customised and modified, as well as being easily shared and used. [22] It is crucial to acknowledge that the

execution of these codes is exclusively possible within the Capella framework, as *Python for Capella* constitutes an internal API.

The figure 3.2 illustrates the architectural levels on which this solution is based. The purpose of the solution is highlighted in red, while the contributions of the users are indicated in blue.



**Figure 3.2:** *Python for Capella* architecture [22].

Among these levels, the **Java API** bridges Python with Java, providing support for data extraction and import. **Python Technical Libraries** simplify operations for users by enabling the definition of high-level scripts. They provide libraries, including the definition of the Capella metamodel. Lastly, **High-Level User Scripts** define the environment within which users implement specific actions to be performed, with default examples provided to demonstrate the capabilities.

*Python for Capella* is supplied with default libraries and relies on a simplified meta-model to facilitate the definition of scripts by end-users. Moreover, documentation is generated from this model to know more about its content [25]. This solution is highly customizable, extensible for Capella metamodel add-ons, and clearly separates technical scripting from high-level user scripts. In summary, Python for Capella empowers users with programmatic access and automation capabilities to manage, analyze, and exchange data within their Capella models more efficiently.

Starting from the scripts already available in *Python for Capella* environment [22], these were modified and adapted to be able to satisfy the requests of the project. The following codes have been implemented within the Capella environment and utilised for the purpose of integration. In order to obtain a more accurate understanding of them, it is recommended to consult the appendix A, where the complete scripts are available.



***Export\_SF\_MO.py***

This script extracts from Capella all the functions of the layer of interest, in this case the System Analysis level, and exports them into an Excel file, divided into three sheets. The first sheet "All System Functions" comprises all the functions of the level of interest and their characteristics, that include the name, ID, description and the identification of those functions which are leaf functions. The second sheet "Leaf Functions" contains only the leaf functions and their related information. The third sheet "Mission Objectives" contains the leaf functions relating to a root function of the user's choice. This layout has been configured with two objectives in mind: firstly, to facilitate the complete export of all functions, and secondly, to enable their import into Valispace.

The configuration of the Excel file is totally customisable. In this particular instance, the script was implemented with the objective of deriving the leaf functions from the root function "To satisfy mission objectives", and subsequently creating new requirements in Valispace, based on the aforementioned functions.

The script is readily adaptable, allowing for its implementation in other Capella's levels and for the generation of a distinct Excel file format.

***Export\_System\_Actor.py***

This script simply extracts all the *System Actors* created at the System Analysis layer on Capella and lists them in Excel format.

***Export\_Logical\_Component.py***

This script extracts all the *Logical Components* defined at the Logical Architecture level on Capella. The extraction process starts with the selection of a component by the user, from which all the components that originate from it are exported. The script reports the component's hierarchy throughout the extraction process.

In order to facilitate the import operation into Valispace, the data is extracted in Excel format as follows: the names of the components are placed in one column, the names of its parent in another, and a number indicating its hierarchy level in a third. If the number of the hierarchy level is 1, the component is at the top of the hierarchy tree; if the number is 2, the component is one level down, and so on.

It is important to note that, with negligible alterations, this code can be executed on different Capella levels, thereby encompassing the components of interest and their respective hierarchies.

### ***Export\_\_modes.py***

The purpose of this script is to enable the exportation of Capella's *Modes*, defined via the MSM diagram, as a list of them in Excel format.

## **3.4 Valispace Python API**

Valispace exposes an open Representational State Transfer (REST) API, which can be queried using Python libraries. Its primary objective is to facilitate the process of interacting with Valispace's data. In this project, the software was employed to develop customised scripts capable of interacting with the platform and with data previously extracted from Capella. This solution is characterised by its simplicity and ease of use.

The initial step involves the installation of the library within the selected Integrated Development Environment (IDE). Subsequent to this, authentication on Valispace is required, which can be achieved through the utilisation of either API tokens generated on Valispace or via Valispace's username and password. It offers a simple and intuitive method of executing Create-Read-Update-Delete (CRUD) operations, which represent the fundamental vocabulary for interacting with any data management system. In particular:

- Create (C) to add new data into Valispace through *POST* Hypertext Transfer Protocol (HTTP) method;
- Read (R) to retrieve existing data from Valispace through *GET* HTTP method;
- Update (U) to modify existing data in Valispace through *PUT/PATCH* HTTP method;
- Delete (D) to remove existing data from Valispace through *DELETE* HTTP method.

The following list comprises the scripts created exploiting this API. The primary function of these codes is to support the import of data previously exported from Capella into Valispace, as well as to enable a comparative analysis of the data defined in the two environments.

### ***Import\_\_Requirements.py***

The present script retrieves the functions associated with the mission objectives and their corresponding descriptions from the Excel file that has been generated by the **Export\_SF\_MO.py** 3.3. These functions are then imported into Valispace as new *Requirements*.

The user is required to specify the Identification (ID) of the *Specification* under

which the new requirement will be allocated. The script will then generate a new *Requirement* with an identification number that is consistent with the nomenclature and in line with the previous ones in terms of numbering.

In order to facilitate this process, it has been determined that the "description" section of the *System Functions* in Capella should include the text of the associated requirement, thus enabling it to be imported automatically and without modification.

### ***Import\_System\_Components.py***

This script imports the list of *System Actors*, that has been previously exported from Capella to an Excel file, through the **Export\_System\_Actor.py** script (3.3), as new *Components* in Valispace.

### ***Import\_LC\_Hierarchy.py***

The present script utilises the export results of the **Export\_Logical\_Component.py** code (3.3), which contains the components with their hierarchy derived from the Capella design, and imports them into Valispace while maintaining consistency. The user is required to specify the ID of the component from which the import process shall begin.

### ***Components\_bridge.py***

This script has been created with the intention of creating a bridge between the two tools, Valispace and Capella, verifying the correctness and consistency of their *Product trees*. In this case, a comparison among the various components, checking also the coherence of their hierarchy, is made at the Logical Architecture layer of Capella.

In particular, the code has been implemented through six main functions: once all the *Components* have been retrieved from the Valispace project through the **get\_all\_project\_components** function, the **get\_valispace\_parent\_child\_hierarchy** function builds the hierarchy by identifying their parent-child relationships. Then, the **get\_excel\_data** function reads the Excel file extracted from Capella, taking all the values and converting them into a string, ensuring that the format is compatible with Valispace's one. The **compare\_lists** function compares the two lists of parent-child relationships, derived from the two tools, and displays a report of the analysis in the console, specifying which components are inconsistent.

Furthermore, considering Capella as the master tool, an automation process has been implemented through the **comps\_to\_import\_to\_Vali** function which, in the event that elements present in Capella are missing in Valispace, automatically

proceeds to insert these missing components into the Valispace environment, while maintaining compliance with Capella.

***Import\_All\_Modelists.py***

The present script, starting from the export of the Capella operative modes through the **Export\_modes.py** script (3.3), imports these as *Modelists* into Valispace and assigns them to the *Component* defined by the user.

# Chapter 4

## Case study

The present chapter is dedicated to the practical implementation of the methodology delineated in the previous chapter, with the objective of validating its efficacy in a space mission context. In order to illustrate this process, a case study is presented, which relates to the design of a small satellite mission, which has been named GeoProfundo. The mission has two objectives: the first concerns the emulation of an interplanetary environment in LEO, and the second is an Earth observation objective. The subsequent paragraphs provide a detailed study of this mission, outlining the phases to be followed in accordance with the integration methodology. The work carried out on Valispace and Capella is also provided, emphasising how the adopted approach ensured compliance and conformity between the two tools.

### 4.1 Mission Overview

This section addresses the objectives of the GeoProfundo mission, which is framed within two domains: deep space exploration and Earth observation.

The first domain represents a technical challenge for small satellites. Due to the reduced costs and rapid development cycles, these satellites are destined to play an increasingly significant role in the future of deep space missions. In the context of the mission, they should serve as cost-effective in-orbit demonstrators of critical technologies for future deep space applications, within the limitations of the LEO environment. In particular, the demonstration should focus on navigation, communication and operations capabilities, which are critical for deep space.

On the other hand, the Earth Observation domain presents the scientific challenge of monitoring the effects of UHI phenomenon in European cities. The UHI phenomenon has been shown to have a significant impact on the thermal

environment of urban areas, causing them to exhibit higher temperatures in comparison to suburban and rural regions. It is caused by the "heatwaves", defined as prolonged periods of extremely high temperatures. [26] [27] Whilst it is true that this service already exists, this can be further supported by small satellites, that can be specifically targeted towards a designated area, with the objective of addressing the specific requirements of the relevant stakeholders and end users.

The mission statement of the Geoprofundo mission has been defined as follows: *"The mission should act as a technology demonstrator in LEO for future deep-space small satellite missions, focusing on navigation, communication and operations technologies. Additionally, the mission should monitor relevant environmental parameters across European urban area providing critical insights into the UHI effect."*

The mission goals that can be derived are: (i) to validate small satellites' technology capabilities for deep-space navigation, communication, and operations in a LEO environment and (ii) to monitor and quantify relevant environmental parameters providing critical insights into the UHI effect. The SoW provides a list of drivers and high-level requirements, which can be consulted in the table 4.1.

**Table 4.1:** High Level Requirements and Drivers

Req.ID	Statement
<b>R-MIS-010</b>	The mission shall support autonomous operation for a minimum of 10 (TBC) days without human intervention.
<b>R-MIS-020</b>	The mission shall simulate Mars communications architecture, achieving a minimum delay of 13 TBC minutes for signal transmission.
<b>R-MIS-030</b>	The mission shall support in-situ mapping with a spatial resolution of at least 100 (TBC) meters.
<b>R-MIS-040</b>	The mission shall be operative with only 1 launch.
<b>R-MIS-050</b>	The mission shall be operative with only 1 launch.
<b>R-MIS-060</b>	The mission shall be compatible with at least 2 commercial launchers.
<b>R-SYS-010</b>	The spacecraft mass shall not exceed 50 kg.
<b>R-SYS-020</b>	The spacecraft shall have propulsion capability onboard. Rationale: towards zero debris policy e.g. Collision Avoidance Manoeuvre (CAM)
<b>R-DES-010</b>	The mission shall be compliant with respect to Space Debris Mitigation Policy of the European Space Agency.
<b>D-MIS-010</b>	The mission shall preferably use European technologies.
<b>D-MIS-020</b>	The Mission should reuse as much as possible existing Earth infrastructures, ground antennas and logistics for operations.
<b>D-MIS-030</b>	Existing off-the-shelf products and technologies for both Ground and Space Segments should be selected when possible.

## 4.2 Stakeholders Needs Analysis






The process starts with the importation of the initial mission and client needs, derived from the SoW and listed in table 4.1, into Valispace as high-level requirements and drivers.

Then, preliminary mission studies concerning the stakeholder needs analysis are conducted on Capella. This analysis is implemented at the first level of the ARCADIA method, known as Operationa Analysis (OA), which is the current focus. As previously stated, this phase entails the identification of the entities, i.e. the stakeholders, who will interact with the system, along with the activities allocated to them and the interactions between these entities.

The primary objective of this study is to identify the needs and objectives of the users. The Operational Analysis in Capella involved the creation of a "domain model" [19], independently of the future system to be realized. The idea is to voluntarily create a level of abstraction from the system under study in order to focus on the "real" needs of the different stakeholders.

Before proceeding to the description of the process, it is first necessary to provide a summary of the terminology employed in this particular context 4.2.

**Table 4.2:** Operational Analysis terminology [19].

Symbol	Element	Description
	Operational Capabilities	Organisation's capacity to provide a service that satisfies the highest level objectives.
	Operational Entity	A real-world entity that interacts with the system and its users.
	Operational Activities	The steps that a process must take in order to achieve a specific objective and which are realised by an entity.
	Operational Interaction	The exchange of information between the various activities.
	Operational Processes	Distinct set of activities and interactions that are interconnected by a common capability.

To proceed with the description of the procedure, the first diagram created on Capella was the OCB diagram 4.1, that allows for the creation of *Operational Capabilities*, *Operational Entities* and the relations between them. Initially, the main stakeholders involved in this mission were identified. A key stakeholder is the European Space Agency (ESA), which can be interested in both the implementation and testing of new technologies in deep space, and in complementing and improving

the service offered by Copernicus on UHI analysis [26] [28]. Furthermore, research bodies and universities are interested in both aspects of the mission, depending on their field of research. This interest is related to the data collected and the goodness of knowledge return. Furthermore, the space industry is to be considered, in terms of companies interested in testing new technologies for deep space and Earth observation. The objectives of these companies are dual: firstly, to increase the Technology Readiness Level (TRL) of their products, and secondly, to gain experience.

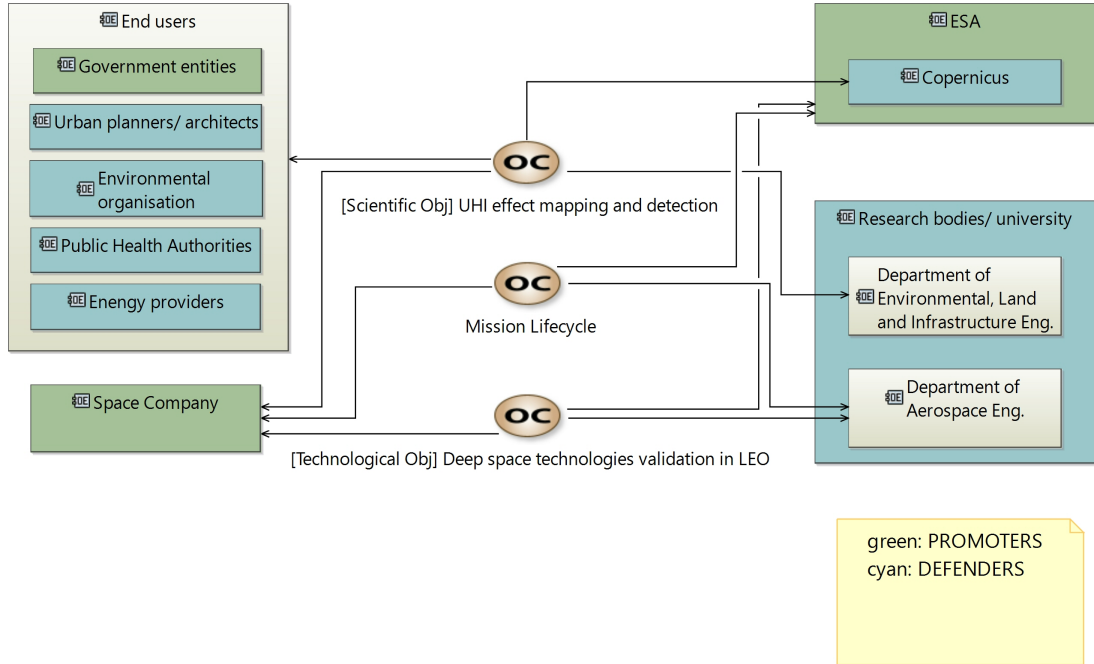
The last *Entity* is associated with end users, who will benefit from the outcomes of the scientific part of the mission. In this case, there are many different entities involved, including national and regional governments seeking to improve the quality of life in cities and to support green urbanism, in the latter case together with architects and urban planners; environmental organisations and public authorities interested in protecting the environment and public health, but also concerned about the effects of climate change. [29][30][31] Finally, there are energy providers who are interested in monitoring the UHI effect in order to make new improvements in the management of increasingly demanding energy.

Despite the potential involvement of numerous other stakeholders in the mission, the individuals with the greatest influence and power are highlighted here. This is illustrated in the legend of the figure 4.1, which differentiates between Promoters, who possess significant interest and influence, and Defenders, who have a high level of interest but less influence.

Consequently, three *Operational Capabilities* were delineated. The first one is the "Mission Lifecycle" and concerns the steps that the satellite design and development process must go through in order to realise the final product. These steps begin with the funding of the mission and end with the release of the satellite into the orbit. The other two *Capabilities* focus on the mission's high-level objectives: one is the scientific objective of "[Scientific Obj] UHI effect mapping and detection", and the other is the technological objective of "[Technological Obj] Deep space technologies validation in LEO". These objectives align with those previously mentioned in the Mission Overview section (4.1) and will be addressed in greater detail subsequently.

Finally, the interactions between *Capabilities* and *Entities* are highlighted in the diagram.





**Figure 4.1:** OCB Operational Capabilities

The following step in the process is the creation of an OAIB diagram for each of the *Operational Capabilities*. The OAIB is a data flow diagram that can be related to each *Capability* and allows for the creation of a set of *Activities*, as well as the *Interactions* that links them. The OAIBs created are shown in figures 4.2, 4.3 and 4.4. They enable a more detailed analysis of the high-level mission objectives at an abstract level, without going into the technical specifics of implementation. In summary, they establish a comprehensive and precise understanding of the system's operational requirements before the implementation process is addressed. For instance, the diagram in the figure 4.3 illustrates the process that the collected environmental data should follow in order to reach the various users. Among the environmental data necessary for measuring the UHI effect, Land Surface Temperature (LST) data and several spectral indices have been selected, including Normalised Difference Vegetation Index (NDVI), Normalised Difference Built-up Index (NDBI) and Modified Normalised Difference Water Index (MNDWI). However, here the focus is on the path that the data will follow to satisfy stakeholders. The same applies to the OAIB Technological Objective diagram in the figure 4.4.

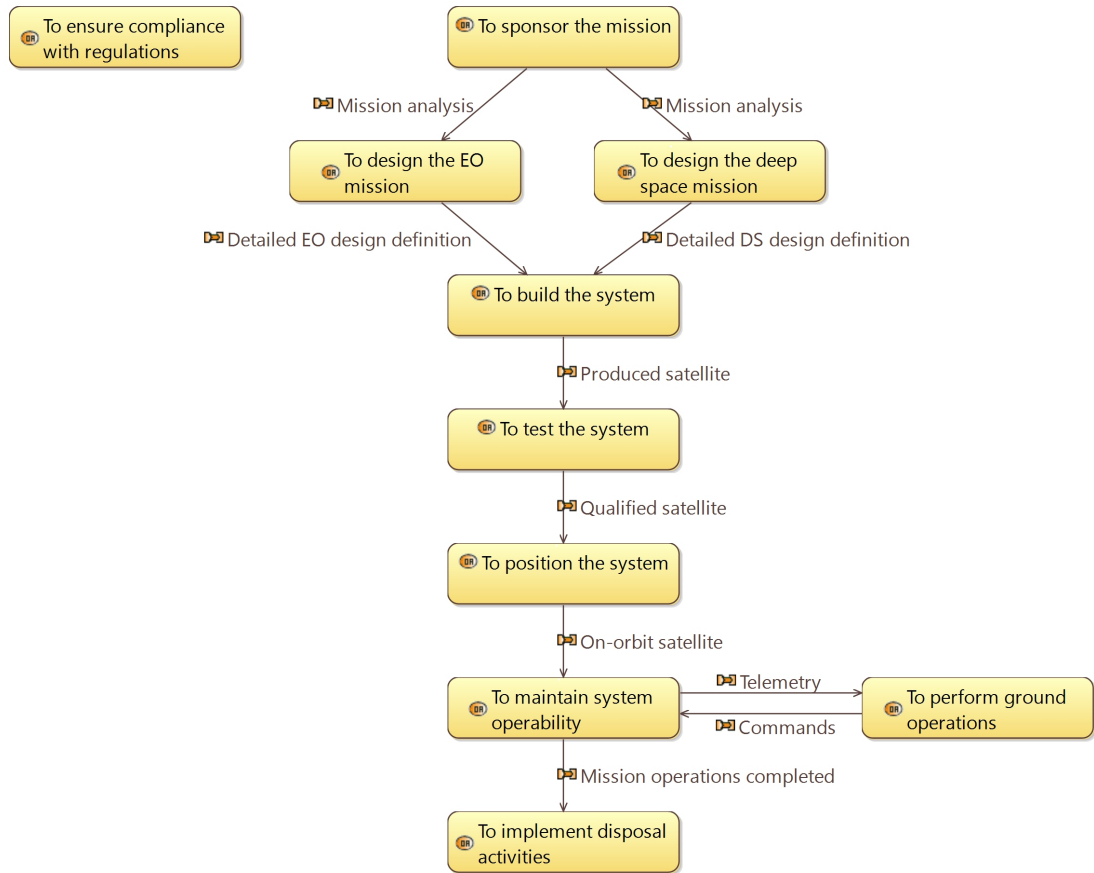


Figure 4.2: OAIB Mission Lifecycle

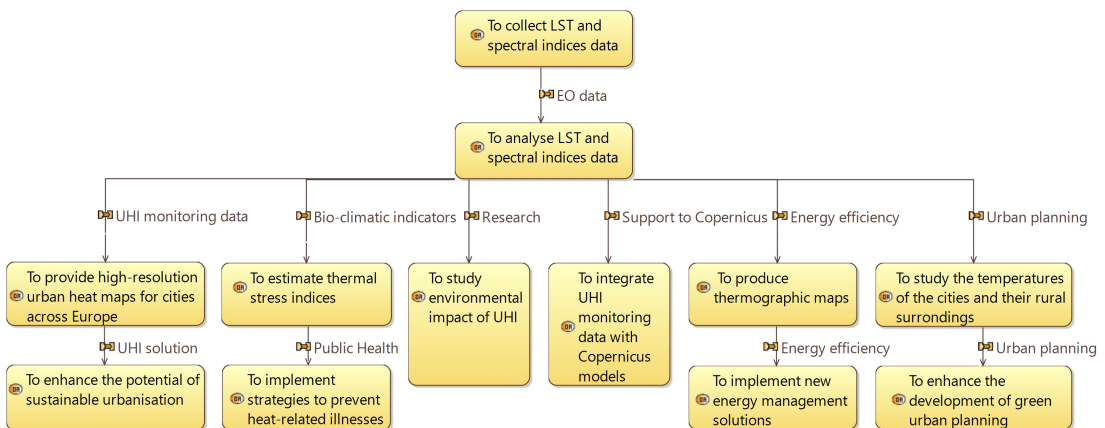
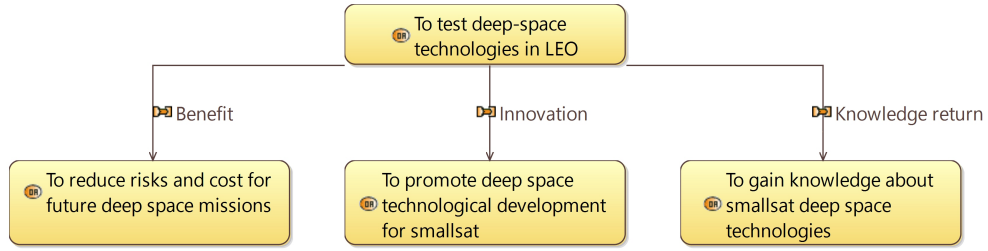


Figure 4.3: OAIB [Scientific Obj] UHI effect mapping and detection



**Figure 4.4:** OAIB [Technological Obj] Deep space technologies validation in LEO

Finally, the OAB architecture diagram was created. It is probably the most important at each level of engineering since it offers the most complete overview of the level. It essentially allows the *Activities*, defined in the OAIB diagrams, to be allocated to each responsible *Entity* that will perform them.

In order to address the complexity involved in developing a space mission, Consortia (i.e. strategic partnerships) are frequently established. These are temporary collaborations among multiple entities with the objective of achieving a common goal, in this case the development of the satellite itself. Initially, roles related to the Consortium were identified and assigned to the relevant entities, as illustrated in the figure 4.5 by the purple rectangles. The definitions under discussion are primarily related to the "Mission Lifecycle" *Capability*, as illustrated by the use of blue arrows to denote the associated *Operational Process*.

The roles in question include that of the Sponsor, in this case the ESA, that is responsible for financing the mission, defining its general objectives and, in this case, also acting as the regulatory entity. Then the role of Principal Investigator (PI) was introduced as the scientific responsible for the mission. In this particular instance, given the dual purpose of the mission, two PIs have been delineated: one related to the scientific objective of the mission, i.e. the Department of Aerospace Engineering of a university, and the other connected to the technical scope, i.e. an aerospace company.

Additionally, a Prime Contractor, named Prime, is involved and is defined as an entity responsible for the overall management of the technical implementation of the product.

The diagram also illustrates the *Operational Processes* associated with the two primary mission objectives identified in this study, which are represented by red and green to denote scientific and technological *Capabilities*, respectively. Consequently, all interactions between the entities and the functions assigned to them are clearly visible.

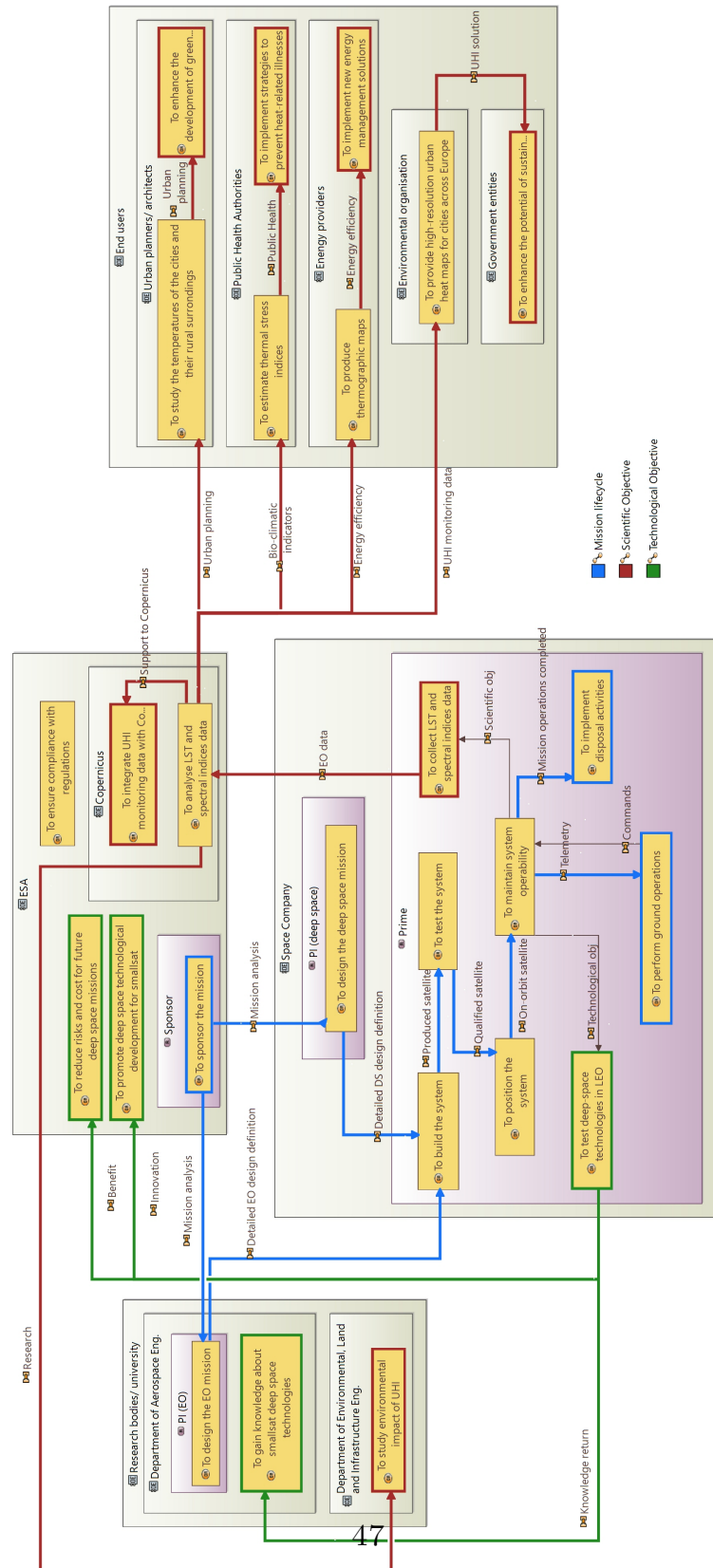
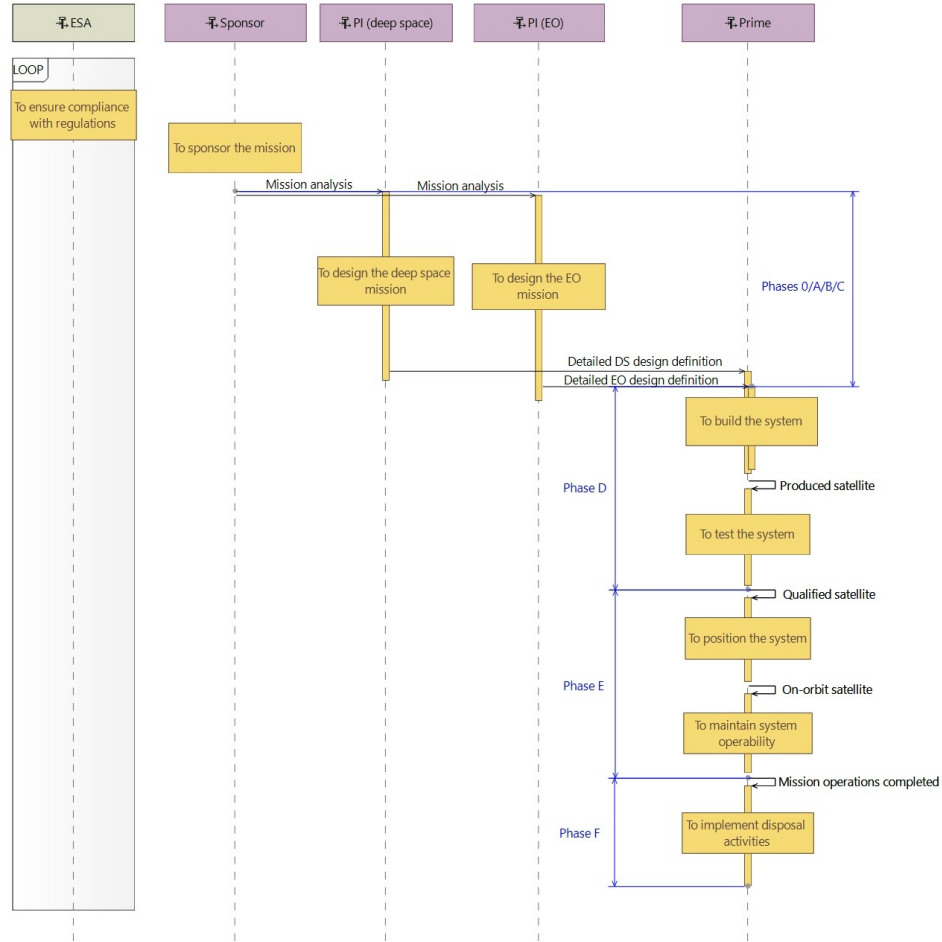


Figure 4.5: OAB Operational Entities

Finally, an *OES* diagram shows the process represented in the OAIB or in the OAB diagrams, but focusing on the chronological aspects, thus describing the sequence of interactions in time. In this case, an OES diagram for the "Mission Lifecycle" *Capability* has been developed and it is shown in the figure 4.6. LOOP fragments in it indicate that the process represented is repeated every day.



**Figure 4.6:** OES Mission Lifecycle








Subsequent to the completion of the Operational Analysis on Capella, the process returns to Valispace, where any newly identified mission requirements will be integrated and the analysis conducted on Capella will be verified as a means of ensuring compliance with the requirements imposed on Valispace.

### 4.3 System Analysis

The focus now shifts to the System Analysis, which is initially implemented at Capella's System Analysis level. In this case, the emphasis is directed towards the system itself, defining its functionality and its external interfaces. This is achieved through a high-level functional analysis and a definition of the *Actors* involved. These actions will subsequently result in the definition of the ConOps and the mission architecture.

As in the previous section, the table 4.3 provides a concise summary of the main Capella's concepts that are highlighted in this section.

**Table 4.3:** System Analysis terminology [19].

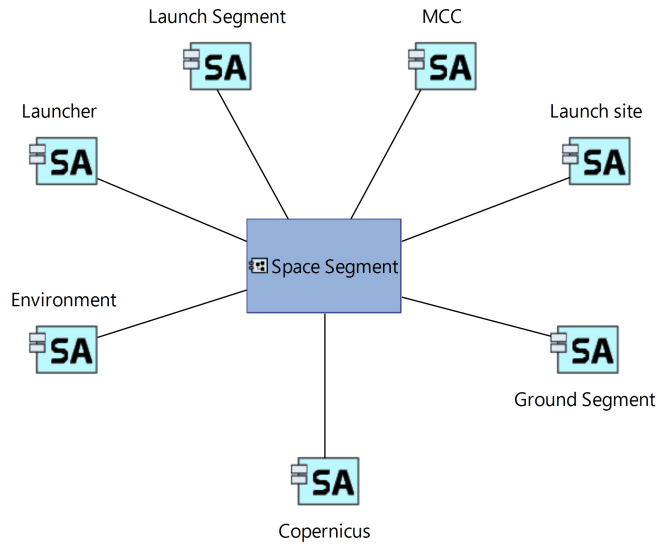
Symbol	Element	Description
	System	A group of elements that function as a single entity ("black box") and respond to user needs.
	System Actor	An entity external to the <i>System</i> (whether human or not) with which the <i>System</i> itself interacts.
	System Capability	The <i>System's</i> capacity to provide a service that enables the accomplishment of high-level objectives.
	System Function	A function accomplished by the <i>System</i> itself, or by an <i>Actor</i> , constituting a behaviour of the system.
	Functional Exchange	A unidirectional exchange of information between two <i>Functions</i> .
	Component Exchange	An interaction between the <i>System</i> and the <i>Actors</i> .
	Functional Chain	A specific pathway among all possible functional paths, which is useful for the identification of constraints.

As outlined in the manual [19] included in the bibliography, there are two methods for transitioning from the Operational Analysis to the System Analysis level on Capella. One method uses the "transition" function to move *Operational Entities*, *Capabilities* and *Functions* directly from one level to another. The other solution, which has been chosen for this case study, keeps the two levels separate, even if a certain continuity is noticeable. In the System Analysis level, the focus is indeed on defining high-level *System Functions* that are no longer associated with the role of the stakeholders. The *System* is identified as a modeling element, i.e. a "black box" containing no other structural elements. [19]

The design process starts with the CSA diagram in figure 4.7, which is one of the

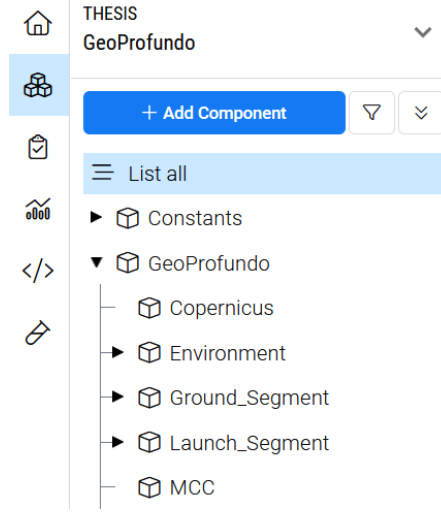
first diagrams to be realised in the System Analysis on Capella. It defines the System of Interest (SoI), namely the "Space Segment", that, in this case study, is the GeoProfundo satellite, and new *System Actors*.

Some of the *Operational Entities* previously identified at OA have been transitioned to System Analysis (SA). The definitive set of *System Actors* comprises the Launch Site and Launcher, which belong to the Launch Segment; the Mission Control Center (MCC), which plans operations; the Ground Segment, which manages ground operations; Copernicus, which integrates the collected environmental data with its models; and finally an entity called Environment, which embodies space environmental behaviour.



**Figure 4.7:** CSA System

These *Actors* defined on Capella were then imported as *Components* into Valispace using the two aforementioned scripts, **Export\_System\_Actor.py** (3.3) and **Import\_System\_Components.py** (3.4). This ensures consistency between the two tools at an early stage, avoiding errors later on. These new *System Components* are included in the System Design section of Valispace, under the GeoProfundo Component. The figure 4.8 below shows the results of the import process.



**Figure 4.8:** System Components imported from Capella to Valispace.

The subsequent part of the process involves the creation of the MCB diagram, which defines *System Missions*, i.e. system objectives, and new *System Capabilities*. Moreover, it identifies the *System Actors* involved in each *System Capability*.

In this instance, a series of *System Capabilities* are derived from a generic *Mission Capability*, thereby establishing the foundations upon which the subsequent functional analysis is to be conducted. The defined *System Capabilities* can be observed in the figure 4.9, where they are also related to the *Actors* through relationships labelled "Capability Involvements".

Subsequently, the decomposition of *System Capabilities* is conducted, thereby leading to the realisation of a rigorous functional analysis through the SFBD diagram, that allows the creation of functions and sub-functions. It is significant to observe that the *Functions* indicated in light blue are those that have been allocated to the *Actors*. As recommended in the manual [19], it is considered a best practice to modify the color of the parent functions that can no longer be allocated to white. Consequently, the only functions, namely leaf functions, displayed in green, are those allocated to the *System*. The figures 4.11 and 4.10 show snapshots of the conducted functional analysis. The other SFBDs diagram, related to the remaining *System Capabilities*, can be consulted in the appendix C.



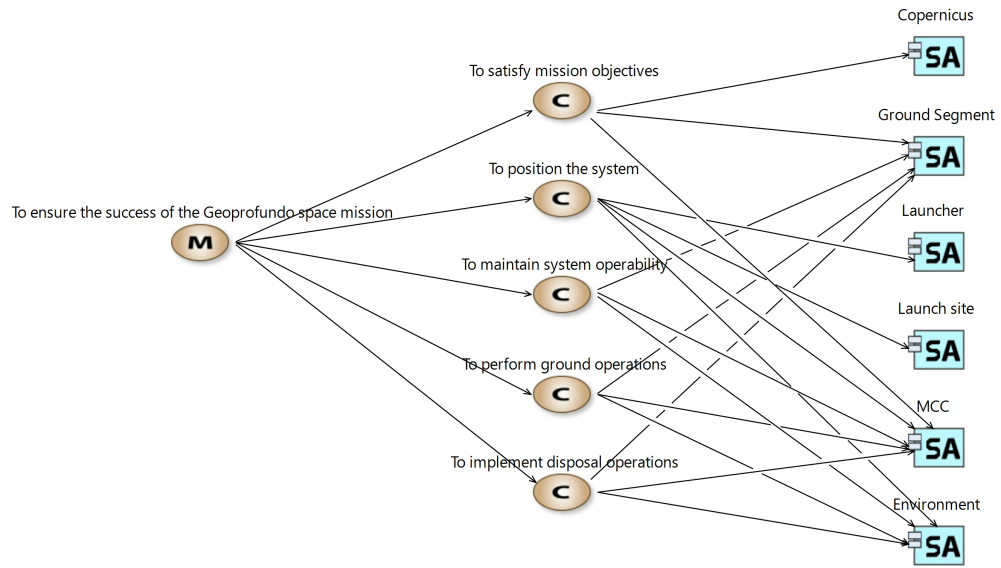


Figure 4.9: MCB Mission and Capabilities Blank diagram

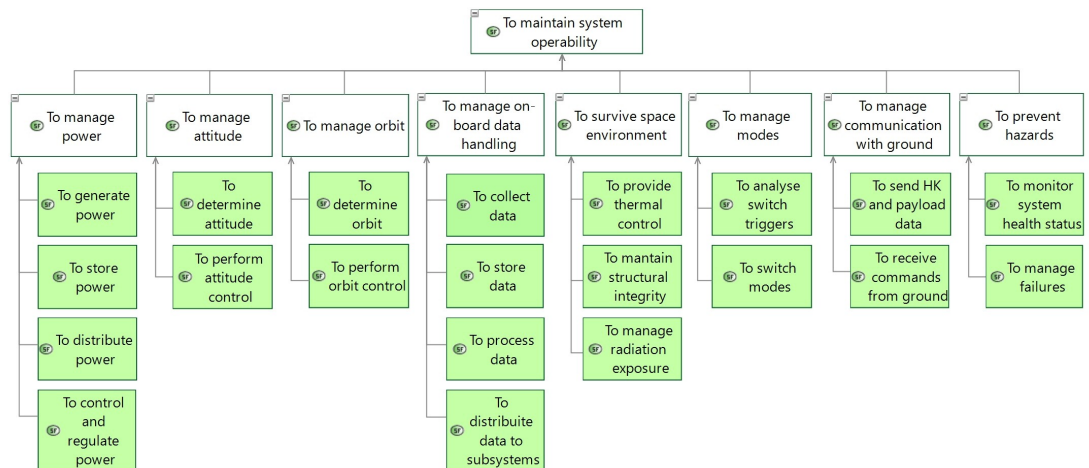
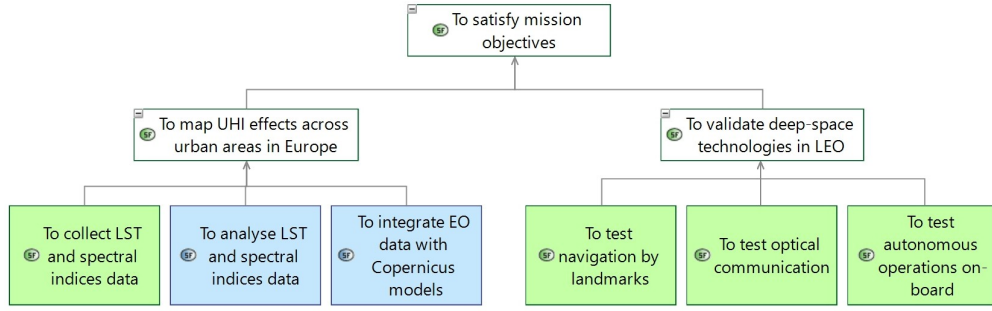


Figure 4.10: SFBF To maintain system operability



**Figure 4.11:** SFBBD To satisfy mission objectives

Once the initial preliminary functional analysis has been conducted on Capella, the mission requirements on Valispace have been updated. By employing the **Export\_SF\_MO.py** (3.3) and **Import\_Requirements.py** (3.4) scripts, which respectively export the leaf functions of the selected root function from Capella and then import them as new requirements on Valispace, Valispace is maintained in compliance with the initial design choices implemented on Capella. In order to execute the code, the user is required to submit the Capella root function of interest, in this case "To satisfy mission objectives", along with the Specification ID on which the requirements will be imported into Valispace. Consequently, the script enables the generation of new requirements with an identifier that is consistent with the adopted convention, with a consecutive index with respect to the last requirement in the selected specification.

The figure 4.12 presents a screenshot of the Valispace platform, which illustrates the results of this process. In this particular example, the requirements refer to the testing of deep space technologies and they have been imported under the "MISSION" Specification. The technologies selected for this mission encompass the validation of navigation by landmarks, the testing of optical communication, and the automation of certain on-board operations.

In any case, this constitutes one potential method for directly creating requirements on Valispace from a *System Function* on Capella. The process could also be performed by creating a requirement on Capella and importing it into Valispace, or by managing the requirements manually in the Valispace environment.

Identifier ↑	Text
<input type="checkbox"/> <b>R-MIS-100</b>	The mission shall perform optical communication testing.
<input type="checkbox"/> <b>R-MIS-110</b>	The mission shall perform autonomous operation testing.
<input type="checkbox"/> <b>R-MIS-120</b>	The mission shall perform navigation by landmark testing.

**Figure 4.12:** Mission Requirements imported from Capella to Valispace.

Returning to the Capella environment, the SDFB has been created. This is a data flow diagram that defines *Functional Exchanges* and allocates them to the *Functions*. The functions constituting a *Capabilities* can be organised using these diagrams, as shown in figure 4.13. The same type of diagram has also been developed to illustrate *Functional Exchanges* involving multiple *Capabilities*, interconnected by a shared concept, in this case, the data flow. The figure 4.14 illustrates the functions involved and their mutual interactions.

It is possible to observe all of the *System Functions* involved in this level in the *SAB* diagram, as shown in figure 4.16 where the *Functions* are allocated to the *System* or to the *Actors*. In this instance, a global diagram has been chosen; however, it can also be partial, enabling the user to select the entities to be displayed. Additionally, it introduces the concept of *Component Exchange*, that, according to the ARCADIA method, represents an exchange that crosses the *System* or an *Actor* boundary. As previously stated, the System Analysis does not incorporate the concept of subsystems. However, examining the diagram in figure 4.16, a graphical representation of the functions that belong to each subsystem could be gathered.

Another feature offered by Capella exploited in this level is the definition of the ConOps. In order to implement this functionality, it is necessary to exploit the VPMS add-on , which offers the capability to create the "Configurations", that can be associated with the phases of the ConOps. The "Configurations" determined in this case have been I&T, LEOP, the Operative Phase and the

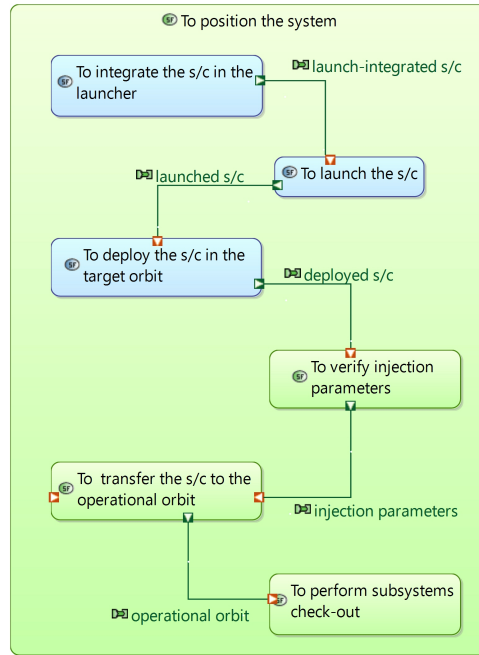


Figure 4.13: SDFB To reach the operational orbit

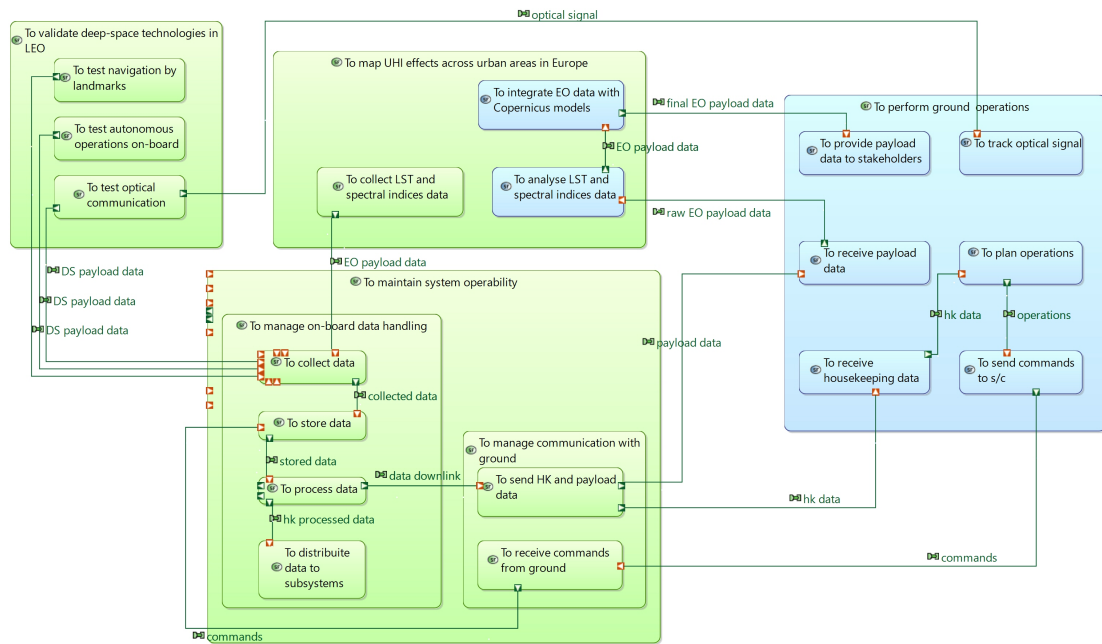
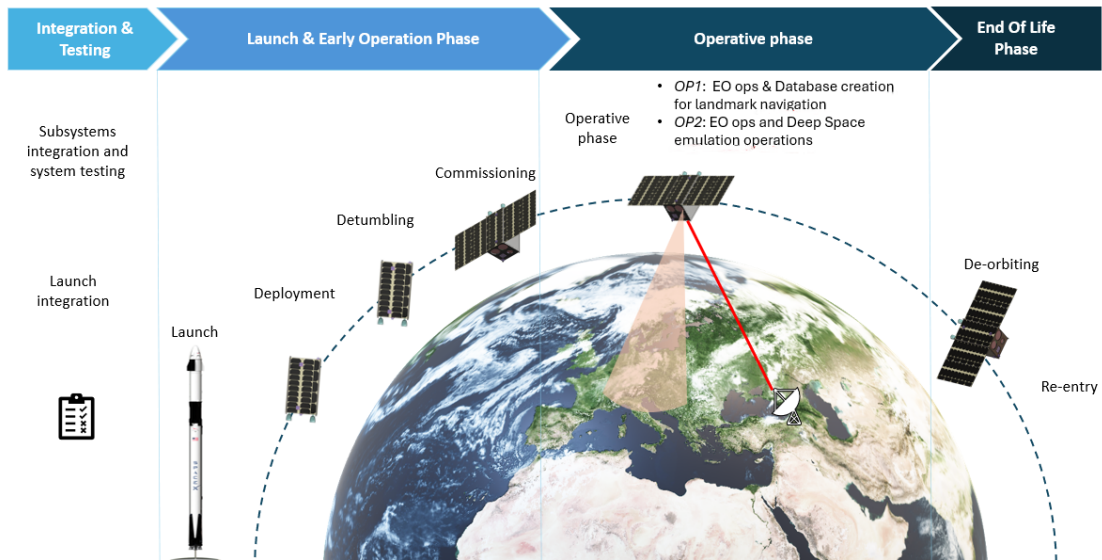


Figure 4.14: SDFB Data flow

EOL. The initial I&T phase concerns the integration and testing operations conducted prior to launch, while the LEOP encompasses the launch, the release into the target orbit, the deployment, the detumbling, and the commissioning phases. The Operational Phase is comprised of two distinct sub-phases. During the first sub-phase, environmental data is collected and a database on Earth mapping is created for the purpose of implementing navigation by landmark. In the subsequent sub-phase, while the collection of environmental parameters for Earth Observation (EO) purposes is continued, the deep space technologies are tested. Finally, the EOL phase involves de-orbiting, passivation and re-entry. The figure 4.15 shows a visual representation of ConOps.



**Figure 4.15:** ConOps visual.

In the context of Capella, these phases are modelled by indicating, for each configuration, which *System Functions* are active and which are not, as illustrated in the figure 4.17 for the Operative configuration. The operational status of each function is denoted by a specific colour: green indicates that the function is active, while red signifies that it is inactive. The remaining configurations are delineated in the appendix C.

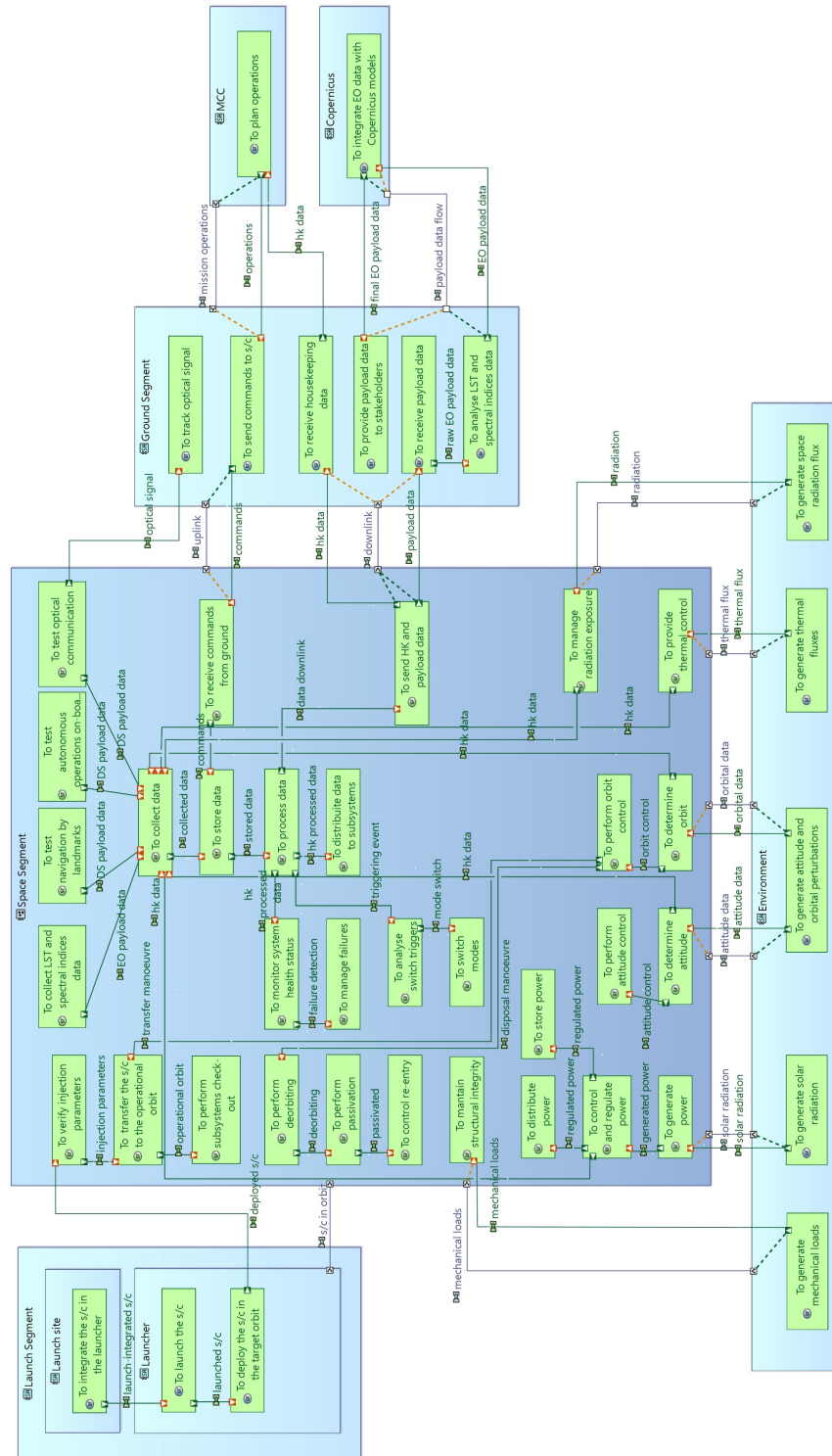


Figure 4.16: SAB Structure








## 4.4 System Design

Once the high-level functions and architecture have been defined in the System Analysis, the following step is to understand how the system works to meet the objectives and then open the so-called "black box". This procedure is denoted in Capella as "Logical Architecture" and involves a further decomposition of the functions of the previous level, as well as a breakdown of the *System* into *Logical Components*, assimilated to the concept of subsystems. It is important to note that the present analysis does not encompass technical considerations and implementation choices, which will be addressed in the next level.

Similarly as in the previous sections, the table 4.4 provides a brief summary of the logical architecture terminology and elements employed in this layer.

**Table 4.4:** Logical Architecture terminology [19].

Symbol	Element	Description
	Logical Component	Structural element within the <i>System</i> , with ports to interact with other <i>Logical Components</i> or <i>Actors</i> .
	Logical Actor	An external entity to the <i>System</i> (human or non-human) with which the <i>System</i> itself interacts.
	Logical Function	Behaviour or service provided by <i>Logical Components</i> or <i>Actors</i> that can communicate with other functions through the <i>Functional Exchanges</i> .

The initial step consists in the transition of the *System Functions* derived from the System Analysis to the current level. Thereafter, a review of these functions has been executed, with the majority of them being broken down. This breakdown specifically impacted those functions derived from the root functions "To maintain system operability" and "To satisfy mission objectives". The results are visible in the LFB (figures 4.18 and 4.19), which allow for the hierarchical identification of the new *Logical Functions*.

The system's approach to testing deep space technologies in the various areas of interest is now clearer. The steps that led to the testing of navigation by landmarks and laser communication are noted. As for autonomous operations on board, these are to be managed through the Failure Detection, Isolation and Recovery (FDIR) algorithm, that will help the transition between operating modes, with commands from the ground being kept to a minimum. It is also important to note that the *System Functions* derived from the previous level and no longer allocated to any entity in this level have been coloured white.



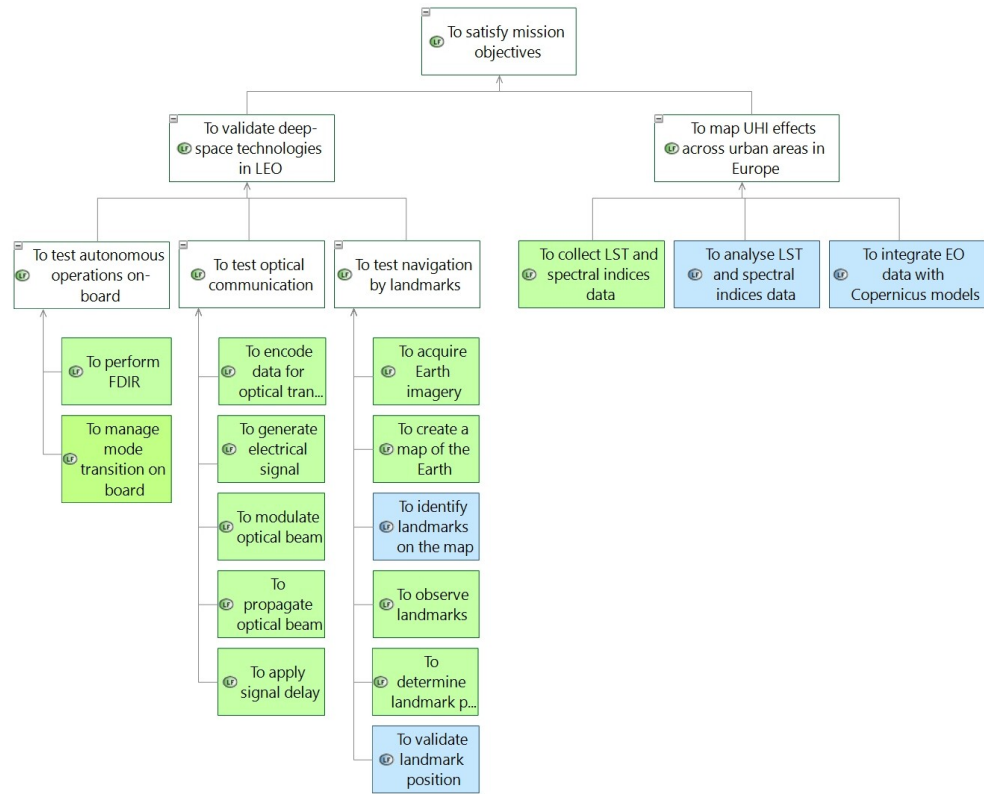


Figure 4.18: LFBD To satisfy mission objectives

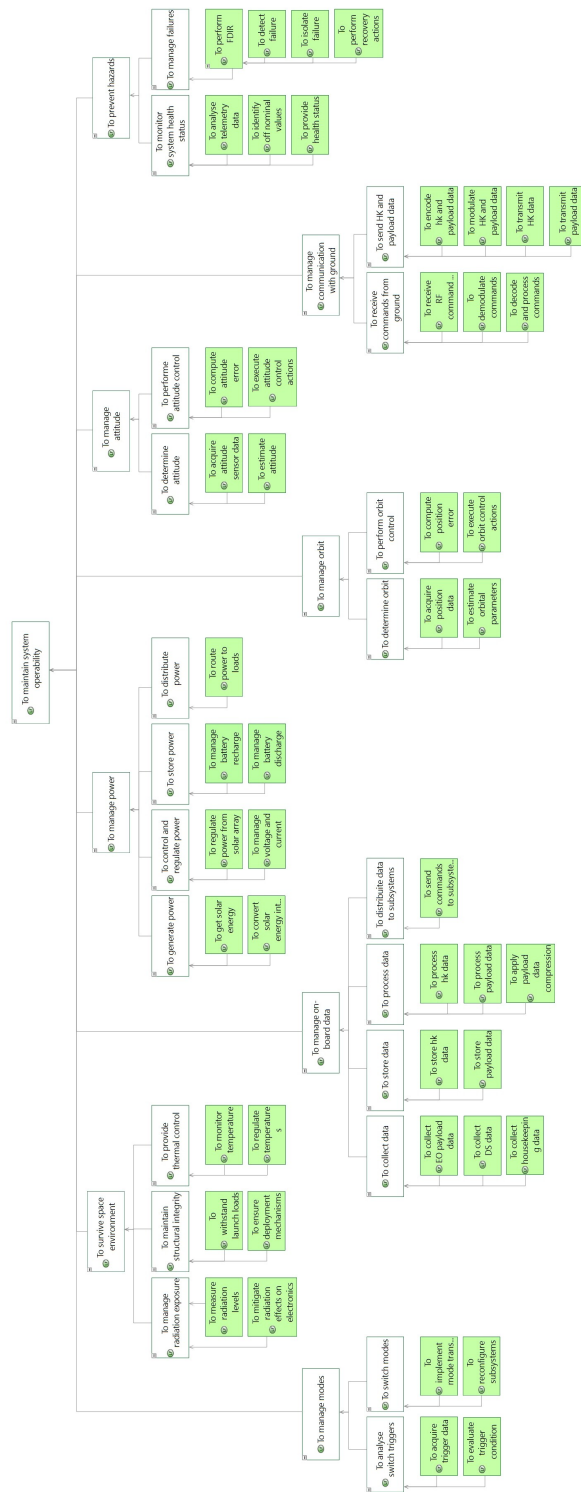
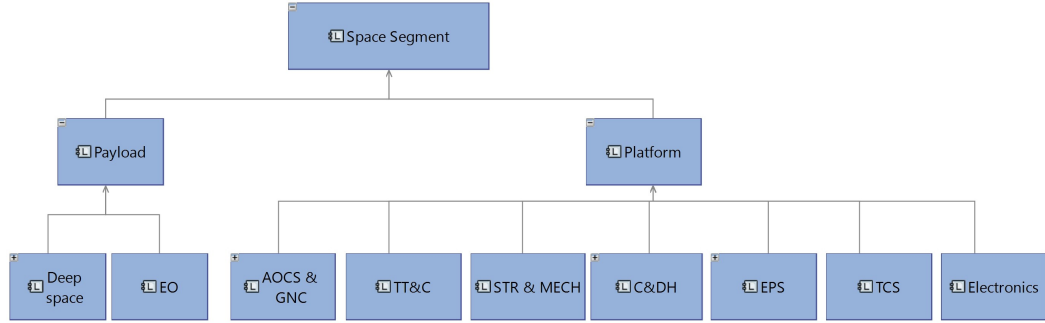


Figure 4.19: LFBD To maintain system operability

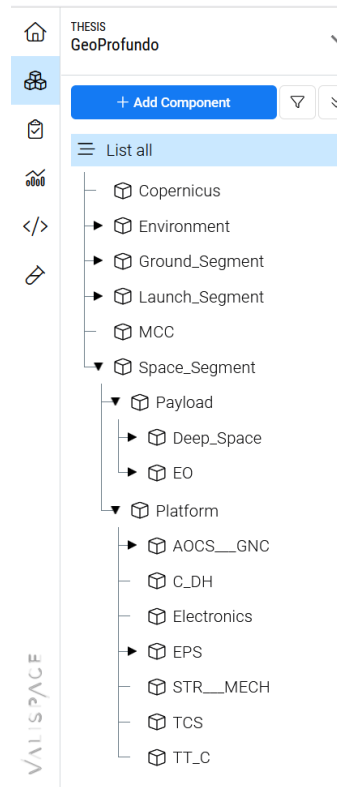
Subsequently, the logical structure has been modelled. The *System* is thus broken down into a series of *Logical Components*, thereby creating a real hierarchy, as illustrated in the LCBD diagram in figure 4.20.



**Figure 4.20:** LCBD Space Segment

As soon as the logical structure of the *System* has been completed, all the *Logical Components* created are imported into Valispace, maintaining the hierarchy and, therefore, the parent-child relationships. The scripts that implement this step are firstly **Export\_Logical\_Components.py** (3.3), which exports the *Logical Components* from Capella to an Excel file, and then **Import\_LC\_Hierarchy.py** (3.4), which imports the Excel file into the Valispace environment and, in this case, under the "Space Segment" component. The result of the import process is illustrated in the following figure 4.21. As clearly shown, continuity and consistency are preserved.

At this stage in the design process, it was deemed necessary to create a link, or "bridge", that would involve both tools simultaneously in order to verify their compliance. The exploitation of this instrument during the design reviews is proposed, for instance, to facilitate the verification of the consistency of components between the tools [20]. The **Component\_bridge.py** (3.4) code is capable of implementing this process, exporting the list of components present on Valispace and Capella, comparing them, and displaying a report of the current status. This report indicates which components are present on Valispace but not on Capella, and vice versa. Furthermore, given the adoption of Capella as the "master" tool of the project, in the event that a component is not present on Valispace, the structure is automatically updated in Valispace, thereby restoring conformity. The figure 4.22 shows the output that the code displays in the console when a component is absent on Valispace.



**Figure 4.21:** Logical Component imported from Capella to Valispace.

```
Components that are PRESENT in Capella but NOT in Valispace:
[{'name': 'TCS', 'parent': 'Platform'}]

Comparison completed.

--- Import to Valispace ---
SUCCESS: Parent 'Platform' with 31043 was found.
SUCCESS: Component 'TCS' was imported. Valispace ID: 31113
```

**Figure 4.22:** `Component__bridge.py` output in console when TCS component is missing in Valispace.

Following the identification of the new components and functions, the architecture is modelled, allocating *Logical Functions* to designated *Logical Components* or *Actors*. New *Functional* and *Component Exchanges* are also created. The outcome of the logical architecture is showcased in the LAB diagram, displayed in figure 4.23, which is the result of design choices and decisions involving all the subsystems.

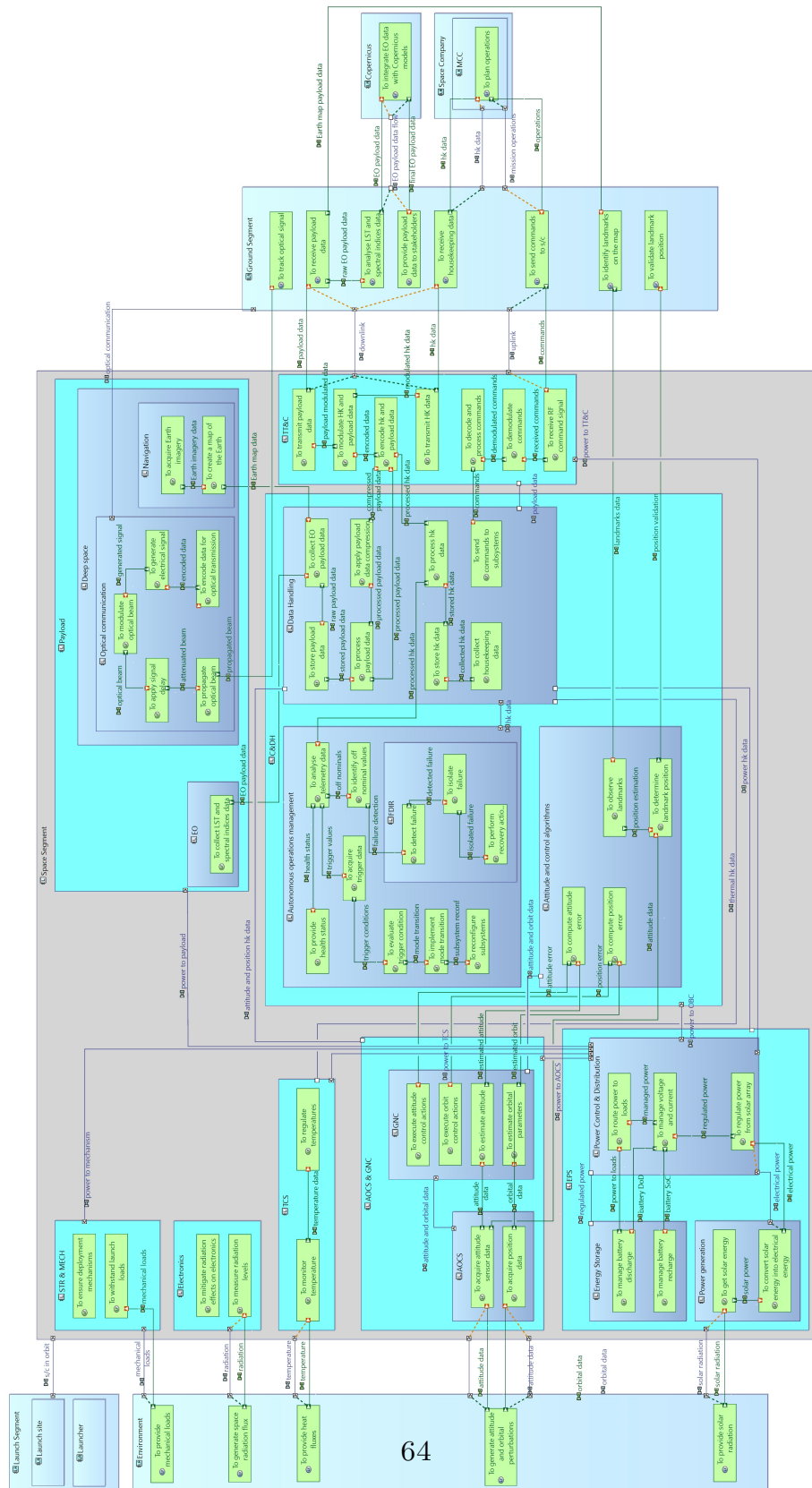


Figure 4.23: LAB Structure

Another potential feature that can be exploited on Capella is the use of MSM diagrams to define operative modes. Modes are defined in the manual [19] as the "expected behaviour", in a specific condition, of an entity (*System*, *Actor*, *Component*). This enables a highly detailed analysis of operative modes through the use of the "transition" functionality, which describes the reaction of a structural element when an event occurs. A transition can be defined as a process which contains a source and a target State, and a Trigger. The diagram employed for this mission is illustrated in the figure 4.24.

In this case, 11 operative modes have been defined: (1) **Dormant**, during which the Spacecraft (s/c) is integrated into the launcher and the entire system is powered off; (2) **In-orbit checkout**, in which the s/c is released into orbit, turned on, and goes through the detumbling, the deployment of solar panels, and the commissioning of all the subsystems; (3) **Basic**, where the satellite is orbiting areas irrelevant to the mission and is therefore in low power mode; (4) **DataBase Creation**, when it is in sunlight and is mapping the Earth in order to test navigation by landmark in the future; (5) **Transmission**, during which the s/c downlinks payload and telemetry data to ground ; (6) **Science EO**, when it captures science data for UHI measurements; (7) **Manoeuvre**, in which it performs station-keeping, transfer and de-orbiting manoeuvres; (8) **Autonomous Deep**, that is activated once the Earth mapping database has been completed and it is dedicated to testing navigation by landmark and autonomous operations on board; (9) **Transmission Deep**, that tests laser communication; (10) **Safe**, that activates in case of failure and recovers either autonomously or via command from ground if necessary; (11) **Passivated**, which is entered once the mission is complete and disposal operations are carried out.

In the figure, four main transitions are delineated: automatic transitions are represented by black, telecommand transitions by blue, failure detection transitions by red, and the recovery transitions by green. Recovery transitions can occur in both automatic and telecommand modality.

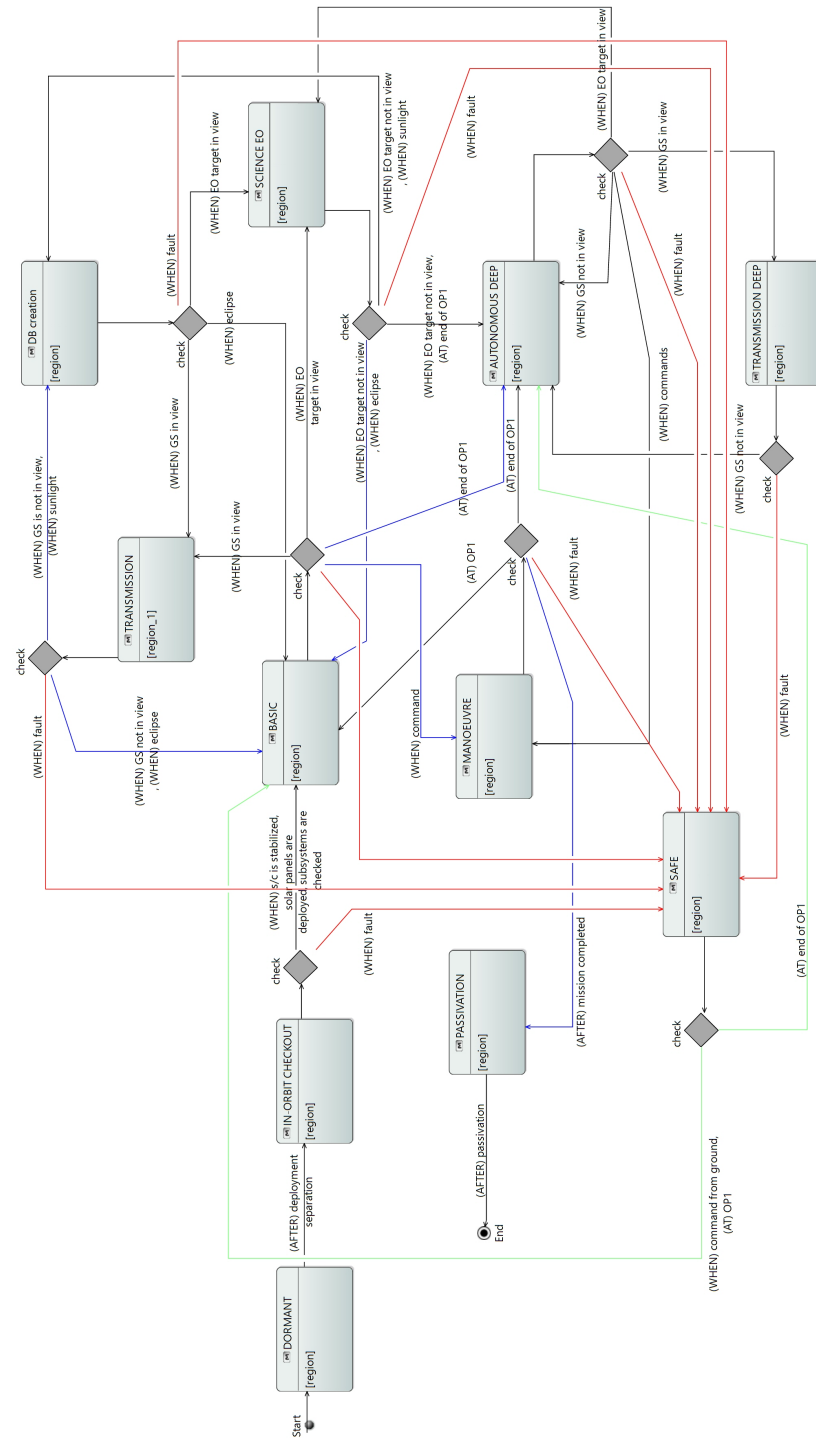
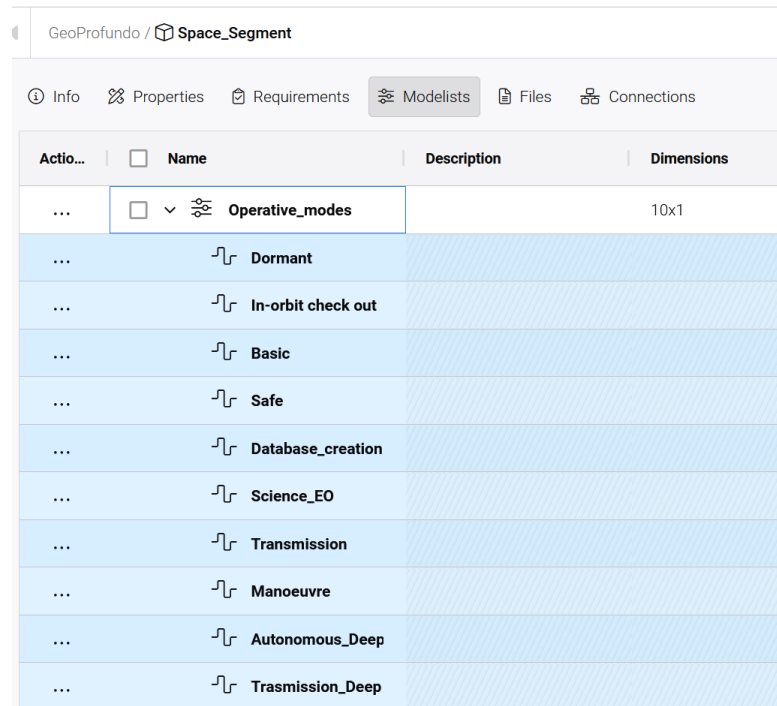


Figure 4.24: MSM Operative modes

The operative modes were subsequently imported into Valispace and associated with the Space Segment, and with the different subsystems. The import process was achieved through the use of the **Export\_modes.py** (3.3) and **Import\_All\_Modelists.py** (3.4), which enable the export of the modes from Capella and the creation of what are defined as *Modelists* on Valispace by indicating the ID of the component in interest. The figure 4.25 illustrates the outcome of the import process into Valispace, in this instance applied to the "Space Segment". The same procedure was applied to the various subsystems.



The screenshot shows the Valispace interface for the 'Space\_Segment' component. The 'Modelists' tab is selected, displaying a table of operative modes. The table has columns for 'Action...', 'Name', 'Description', and 'Dimensions'. The 'Operative\_modes' table is expanded, showing a list of modes with their names and descriptions. The dimensions are listed as '10x1'.

Action...	Name	Description	Dimensions
...	<input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> Operative_modes		10x1
...	<input checked="" type="checkbox"/> Dormant		
...	<input checked="" type="checkbox"/> In-orbit check out		
...	<input checked="" type="checkbox"/> Basic		
...	<input checked="" type="checkbox"/> Safe		
...	<input checked="" type="checkbox"/> Database_creation		
...	<input checked="" type="checkbox"/> Science_EO		
...	<input checked="" type="checkbox"/> Transmission		
...	<input checked="" type="checkbox"/> Manoeuvre		
...	<input checked="" type="checkbox"/> Autonomous_Deep		
...	<input checked="" type="checkbox"/> Trasmission_Deep		

**Figure 4.25:** Operative modes imported into Valispace from Capella

Although the operative modes are significantly clearer and more explanatory on Capella, the process of importing them into Valispace was useful in order to exploit its capacity for budget calculation. Indeed, within the Analysis section of Valispace, it is possible to create a variety of budgets by taking advantage of the properties defined and associated with the various components. The creation of a "Power" matrix property connected to each subsystem, which defines the different power values depending on the operative mode, enables the tool to automatically compute a power budget (4.26).



Actio...	Name	State	Value	Display Unit
...	<b>PowerConsumption</b>		[0 W; 3.44 W; 1.5 W; 1.2 W; 2 W]	W
...	f(x) <b>Dormant</b>		0	W
...	f(x) <b>In-orbit check out</b>		3.44	W
...	f(x) <b>Basic</b>		1.5	W
...	f(x) <b>Safe</b>		1.2	W
...	f(x) <b>Database_creation</b>		2	W
...	f(x) <b>Science_EO</b>		5	W
...	f(x) <b>Transmission</b>		5	W
...	f(x) <b>Manoeuvre</b>		19	W
...	f(x) <b>Autonomous_Deep</b>		2	W
...	f(x) <b>Trasmission_Deep</b>		8	W

**Figure 4.26:** "Power" property associated with AOCS&GNC component, depending on the operative modes.

The results can be viewed both in table form, as shown in the figure 4.27, and in report form. Furthermore, it introduces the possibility of customising the budget through the application of margins, for instance.

In-orbit check out							
Component	PowerConsumption	margin +	total margin +	worst case +	margin -	total margin -	worst case -
▼ Space_Segment	11.14 W	+20%	+27.9%	14.243 W	-0%	0%	11.14 W
▼ Payload	0 W	+0%	+0%	0 W	-0%	0%	0 W
> Deep_Space	0 W	+5%	+0%	0 W	-0%	0%	0 W
> EO	0 W	+5%	+0%	0 W	-0%	0%	0 W
▼ Platform	0 W	+0%	+0%	0 W	-0%	0%	0 W
> AOCS__GNC	3.44 W	+10%	+10%	3.784 W	-0%	0%	3.44 W
— C_DH	2 W	+5%	+5%	2.1 W	-0%	0%	2 W
— Electronics	N/A	N/A	N/A	N/A	N/A	N/A	N/A
> EPS	2.7 W	+5%	+5%	2.835 W	-0%	0%	2.7 W
— STR__MECH	N/A	N/A	N/A	N/A	N/A	N/A	N/A
— TCS	N/A	N/A	N/A	N/A	N/A	N/A	N/A
— TT_C	3 W	+5%	+5%	3.15 W	-0%	0%	3 W
Total	11.14 W	+20%	+27.9%	14.243 W	-0%	0%	11.14 W

**Figure 4.27:** Power budget estimation on Valispace for **In-orbit check-out** mode.

Additionally, the Valispace feature was employed to compute a preliminary mass and volume budget, employing the "mass" and "volume" properties associated with each subsystem. The results can be seen in the appendix C. It is evident that this procedure may also be conducted on physical components, which are to be delineated in the subsequent phase of the project, thus performing increasingly complex and detailed analyses.





## 4.5 Subsystem Design

The next step in the methodology is to achieve a further level of detail by analysing the various subsystems that constitute the satellite. In this case, the process has only been started to give an hint on how it might proceed, but it has not yet been fully implemented. The focus is on the EPS, which serves as an example to demonstrate the potential of Capella and Valispace at this stage of the project.

As in the previous case, the work has started with Capella at the Physical Architecture level. The objectives of the Physical Architecture are equivalent to those of Logical Architecture. However, Physical Architecture defines the concrete physical components that constitute the system. Consequently, it delineates the definitive architectural configuration of the system, accurately representing how it should be integrated. The focus shifts to the functions associated with technical and implementation choices. Furthermore, the components that perform these functions are also defined. [19]

The following table 4.5 provides a comprehensive list of the new concepts introduced in this chapter [19].

**Table 4.5:** Physical Architecture terminology [19].

Symbol	Element	Description
	Behaviour Physical Component	Physical component that executes the allocated <i>Physical Functions</i> , contributing to the behaviour of the <i>System</i> .
	Node Physical Component	Physical component that provides the material resources for <i>Behaviour Components</i> .
	Physical Port	Non-oriented port that belongs to an <i>Node Component</i> ; the <i>Component Port</i> belongs to a <i>Behaviour Component</i> .
	Physical Link	Non-oriented connection between <i>Node Components</i> .

To begin this new phase starting from Capella, the "transition" feature is used to

transfer all the *Functions* defined at the Logical level.

The next step is to define the *Behaviour* and *Node Components*, which in this case have been represented only for the EPS. Three *Node Components* have been defined: the solar arrays, the Power Control and Distribution Unit (PCDU), which is composed internally of Power Control Unit (PCU) and Power Distribution Unit (PDU), and the battery pack. Each of these components has been then associated with a *Behaviour component*, to which the functions already defined at the Logical level could be allocated. The figure 4.28 shows the PAB diagram, which illustrates all the aforementioned choices. It is also possible to define *Functional* and *Components exchanges*. The *Component Exchanges* were categorised as either relating to data, commands, or power lines.

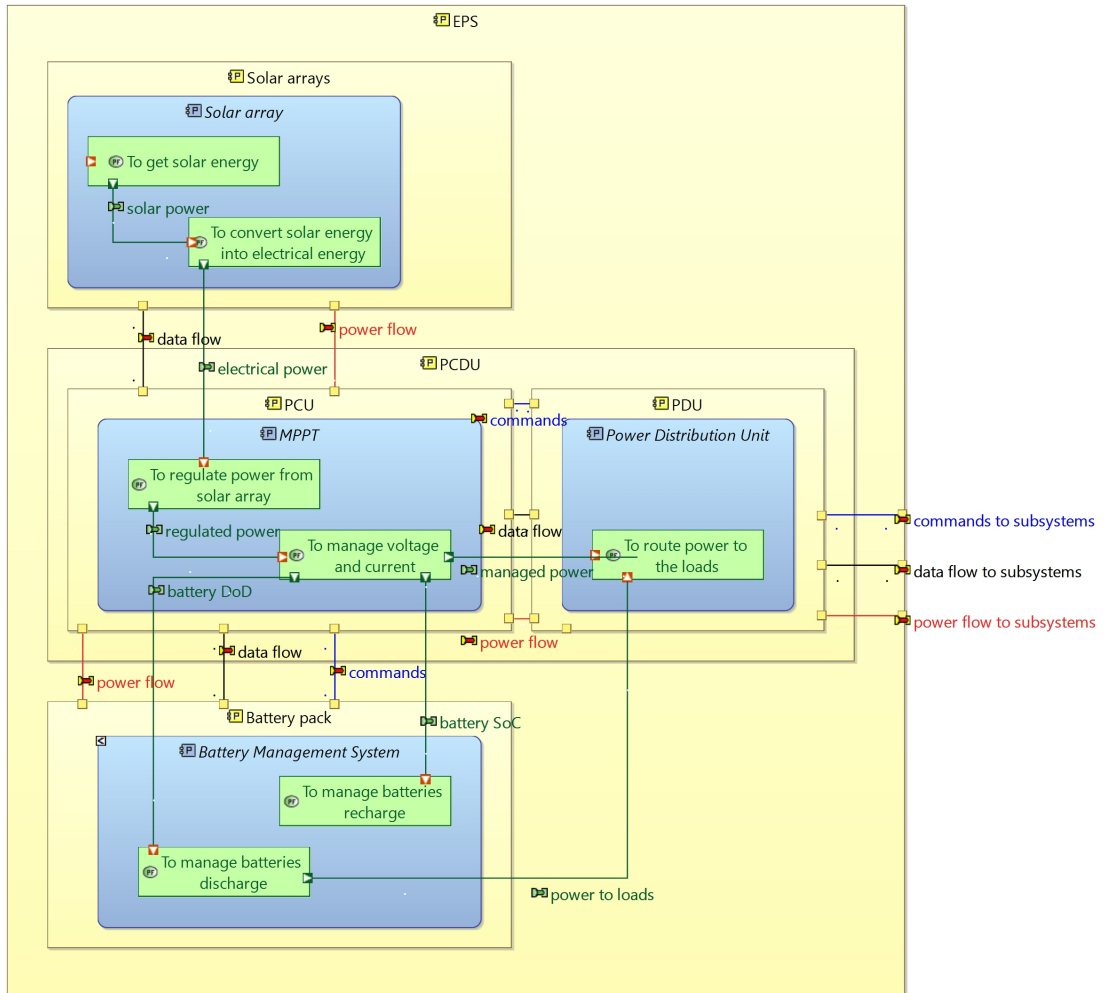
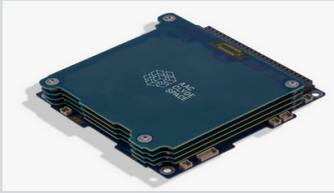
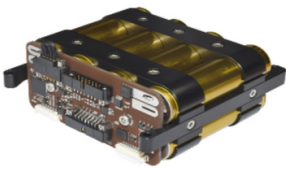


Figure 4.28: PAB Structure EPS

Meanwhile, on Valispace, in addition to continuous requirements management at every level, the components must now be updated. With minor modifications, the codes implemented for the Logical Architecture can be used to bring the new components defined on Capella to Valispace. Again, this is the same iterative process followed in the previous sections, which can also be implemented here using the same Python code, with a few adjustments.

A further capability of Valispace is highlighted: its ability to manage trade-offs between different components. To demonstrate this, an example regarding batteries has been provided in figure 4.29.

	Battery_1	Battery_2
		
Mass	0.67 kg	0.50 kg
Volume_Component	0.49 dm <sup>3</sup>	0.33 dm <sup>3</sup>
Voltage	8.20 V	16.80 V
Batter_Capacity	80.00 Wh	86.00 Wh
	current alternative	select alternative

**Figure 4.29:** Battery trade-off on Valispace. [32] [33]

Valispace provides a clear overview of the components involved in the trade-off by displaying all engineering information in a format that is easy to compare. It also enables to iterate quickly through budgets, as it only requires the selection of one component to update all analyses (for example, mass and volume budgets).

# Conclusions

The present thesis explores the various methodologies employed in the domain of System Engineering to create a more agile and effective approach to the design of small satellite missions. The study was based on two of the most innovative methodologies of recent years: DDSE and MBSE. The complementarity of these two approaches, along with the integration of Capella and Valispace tools, enables the analysis of the mission from different perspectives and throughout the entire life cycle of the product.

Capella has been proven to be the most suitable option for system and subsystem architectures definition and development, ensuring strong consistency and an integrated vision. The ARCADIA method implemented by Capella provides a clear and guided path through the different phases of system engineering, thus helping to structure the design process. Additionally, it facilitates the conception of ConOps and operative modes. In summary, Capella enhances efficiency, quality and reliability in the development of complex systems, providing a structured model-based approach that is essential for addressing the challenges of modern systems engineering.

In parallel, Valispace has been demonstrated to excel in the management of dynamic requirements and engineering data in real-time, facilitating collaborative interactions among diverse teams. Furthermore, the software is designed to estimate budgets and monitor their progression in accordance with the evolution of the design, thereby ensuring accurate tracking of alterations.

However, the true strength of this methodology lays in the integration of the two tools, which enables the bridging of the gaps between them, resulting in the creation of a unified, promising tool capable of completely managing all the early stages of the mission. The seamless interaction between these two tools has been made possible by their capacity to offer APIs, which facilitate not only their own integration but also integration with other platforms.

It is relevant to note that the present study has been conducted exclusively on the preliminary stages of the project, with the intention of exploring

the potential applications of this work. Additionally, it could also function as a starting point for subsequent iterations and increasingly detailed levels, supporting the mission design project in the definition of the physical architecture and the consolidation of the final design. With regard to the design of subsystems, which has only been outlined here, there are numerous possibilities for enhancing the integration of the two tools. For instance, it is possible to associate metadata with components on Capella and then transfer them to Valispace, thus creating new and increasingly realistic budgets. Furthermore, the potential for its application in the domain of Assembly, Integration, Verification and Testing (AIV/T) management is promising, especially if Capella is used in combination with the *Test* section in Valispace.

Other areas that may be worthy of future investigation include requirements management. Valispace has demonstrated a proven expertise in requirements management, which influenced the decision to utilise this platform for management purposes exclusively. An alternative strategy could be to formulate some relevant requirements in Capella before proceeding to import them into Valispace. This would replace the existing process of going through the functions defined in Capella. In this particular instance, the potential disadvantages may be associated with the replication of information across the two software programs. Nevertheless, this remains an alternative that is worth of further consideration.

In summary, the project appears to offer a wide range of possibilities and establishes the foundations for future research into the integration of these two tools.

It appears that Valispace and Capella are complementary in nature, and the integration of these platforms via their respective APIs, *Valispace Phyton API* and *Python for Capella*, has shown promising results. The APIs facilitate the automation of repetitive and time-consuming workflows and tasks, thereby ensuring data synchronisation and reducing human error. The result is an integrated and cohesive toolchain that is fully customisable by the user through Python scripts. This solution is accessible to users with even limited coding expertise, as both APIs provide example codes that can be utilised as a foundational starting point to acquire competencies. In addition, it is widely acknowledged that the selected programming language, Python, is considered to be among the most accessible for beginners.

There are numerous steps that can be pursued in the future. One of these is related to the difference between the two APIs: Valispace exposes a REST API that facilitates interaction with data via HTTP requests from the Python language. In contrast, *Python for Capella* is an "internal" API that enables direct manipulation of models on Capella through scripts executed within the Capella environment itself. It is therefore reasonable to consider an alternative Capella API, designated *capellambse* [34], which was developed before *Python for Capella*. This API allows

models to be read and written on Capella via Python scripts executed outside the Capella environment. The difference is that the *capellambse* API, in combination with the Valispace API, would enable the creation of an independent bridge that can read data from Capella and integrate it into Valispace without the necessity of launching the Capella application. However, further investigation is required into the functionality of this API, in order to evaluate its ability to match the capabilities of *Python for Capella*.

Finally, a next improvement of the tool is the development of a user interface. At present, the integration procedure necessitates the execution of a series of scripts. While these are designed to be straightforward and intuitive for the user, they are still disconnected and fragmented. It is therefore important to be aware of the process outlined in the chapter 3 dedicated to the integration of the tools. The subsequent step could be the creation of a user interface that simplifies the procedure for users with no prior experience of the integration process. This would minimise the training required to familiarise users with the procedure. The implementation of such a system would result in the establishment of an automated and efficient workflow, thereby enhancing product quality and team productivity.

In conclusion, this approach offers new possibilities in the implementation of the MBSE methodology associated with DDSE, potentially throughout the entire life cycle of a small satellite. It raises the issue of the creation of new tools capable of managing the potential of both Capella and Valispace solutions while maintaining a single source of truth.

# Appendix A

## *Python for Capella* scripts

### A.1 *Export\_SF\_MO.py*

```
1 # include for the Capella modeller API
2 include('workspace://Python4Capella/simplified_api/capella.
   py')
3 if False:
4     from simplified_api.capella import *
5
6 # include to read and write xlsx files
7 from openpyxl import *
8
9 # insert the path of your Capella model
10 aird_path = ''
11
12 model = CapellaModel()
13 model.open(aird_path)
14
15 # enter the System Analysis level
16 se = model.get_system_engineering()
17 sa = se.get_system_analysis()
18 root_function_MO = "To satisfy mission objectives"
19
20 # prepare Excel file export
21 project_name = aird_path[0:(aird_path.index("/", 1) + 1)]
22 project = CapellaPlatform.getProject(project_name)
23 folder = CapellaPlatform.getFolder(project, 'script results'
   )
```



```

24 | xlsx_file_name = CapellaPlatform.getAbsolutePath(folder) + '
    | /' + 'Export_SF_M0.xlsx'
25 |
26 | # create a workbook
27 | workbook = Workbook()
28 |
29 | # write Excel file header
30 | worksheetSF = workbook.active
31 | worksheetSF.title = 'All System functions'
32 | worksheetSF["A1"] = 'SF name'
33 | worksheetSF["B1"] = 'ID'
34 | worksheetSF["C1"] = 'Description'
35 | worksheetSF["D1"] = 'leaf function'
36 |
37 | worksheetLF = workbook.create_sheet("Leaf Functions")
38 | worksheetLF["A1"] = 'Leaf functions name'
39 | worksheetLF["B1"] = 'ID'
40 | worksheetLF["C1"] = 'Description'
41 |
42 | worksheetMO = workbook.create_sheet("Mission objectives")
43 | worksheetMO["A1"] = 'MO functions name'
44 | worksheetMO["B1"] = 'ID'
45 | worksheetMO["C1"] = 'Description'
46 | worksheetMO["D1"] = 'Root function name'
47 | worksheetMO["E1"] = 'Specific Ancestor found'
48 |
49 | # retrieve all the system functions from the model
50 | all_SF = sa.get_all_contents_by_type(SystemFunction)
51 |
52 | # examine the hierarchy between functions
53 | def is_descendant_of_specific_function(current_function,
    | root_function_name):
54 |     parent = current_function.get_container()
55 |     current_path = []
56 |
57 |     while parent:
58 |         if isinstance(parent, SystemFunction):
59 |             current_path.append(parent.get_name())
60 |             if parent.get_name() == root_function_name:
61 |                 return True, parent.get_name()
62 |             parent = parent.get_container()
63 |     return False, "Not Found"
64 |
65 | i = 2
66 | j = 2

```

```

67 k = 2
68
69 for sf in all_SF:
70     worksheetSF["A" + str(i)] = sf.get_name()
71     worksheetSF["B" + str(i)] = sf.get_id()
72     worksheetSF["C" + str(i)] = sf.get_description() if sf.
get_description() else ''
73
74     # determine if it's a leaf function and place 'X'
75     is_leaf_function_marker = ''
76     owned_functions_list = []
77
78     if hasattr(sf, 'get_owned_functions'):
79         owned_functions_list = sf.get_owned_functions()
80
81     if not owned_functions_list:
82         is_leaf_function_marker = 'X'
83         worksheetLF["A" + str(j)] = sf.get_name()
84         worksheetLF["B" + str(j)] = sf.get_id()
85         worksheetLF["C" + str(j)] = sf.get_description()
86         j = j + 1
87         is_descendant, SF_root_name =
is_descendant_of_specific_function(sf, root_function_MO)
88
89         if is_descendant:
90             parent_sf_name = sf.get_container().get_name()
91         if sf.get_container() else "No Parent"
92         worksheetMO["A" + str(k)] = sf.get_name()
93         worksheetMO["B" + str(k)] = sf.get_id()
94         worksheetMO["C" + str(k)] = sf.get_description()
95         worksheetMO["D" + str(k)] = parent_sf_name
96         worksheetMO["E" + str(k)] = SF_root_name
97         k = k + 1
98
99     worksheetSF["D" + str(i)] = is_leaf_function_marker
100     i = i + 1
101
102 # save the Excel file
103 workbook.save(xlsx_file_name)
104
105 print('saving excel file')
106
107 # refresh
108 CapellaPlatform.refresh(folder)
109 print('Script execution finished.')

```

Listing A.1: *Export\_SF\_MO.py* script [22].

## A.2 *Export\_System\_Actor.py*

```

1  # include for the Capella modeller API
2  include('workspace://Python4Capella/simplified_api/capella.
   py')
3  if False:
4      from simplified_api.capella import *
5
6  # include to read and write xlsx files
7  from openpyxl import *
8
9  # insert the path of your Capella model
10 aird_path = ''
11
12 model = CapellaModel()
13 model.open(aird_path)
14
15 # enter the System Analysis level
16 se = model.get_system_engineering()
17 sc = se.get_system_analysis()
18
19 # get all System Actors
20 comps = sc.get_system_component_pkg()
21 actors_list = comps.get_all_contents_by_type(SystemActor)
22
23 # examine System Actors hierarchy
24 def actor_hierarchy(system_actors_list):
25     selected_system_actors = []
26     for actor in system_actors_list:
27         container = actor.get_container()
28         if isinstance(container, SystemComponentPkg):
29             selected_system_actors.append(actor.get_label())
30     return selected_system_actors
31
32 actors_to_export = actor_hierarchy(actors_list)
33
34 # prepare Excel file export
35 project_name = aird_path[0:(aird_path.index("/", 1) + 1)]
36 project = CapellaPlatform.getProject(project_name)

```

```

37 folder = CapellaPlatform.getFolder(project, 'script results'
    )
38 xlsx_file_name = CapellaPlatform.getAbsolutePath(folder) + '
    /' + 'Export_System_Actors.xlsx'
39
40 # create a workbook
41 workbook = Workbook()
42
43 # write Excel file header
44 worksheet = workbook.active
45 worksheet.title = 'System actors'
46 worksheet["A1"] = 'Name'
47
48 i = 2
49
50 for single_actor in actors_to_export:
51     worksheet["A" + str(i)] = single_actor
52     i = i + 1
53
54 # save the Excel file
55 workbook.save(xlsx_file_name)
56
57 # refresh
58 CapellaPlatform.refresh(folder)

```

Listing A.2: *Export\_System\_Actor.py* script [22].

### A.3 *Export\_Logical\_Component.py*

```

1 # name : Export Logical Component
2 # script-type : Python
3 # description : Export Logical Component
4 # popup : enableFor(org.polarsys.capella.core
    .data.capellacore.CapellaElement)
5
6 # include for the Capella modeller API
7 include('workspace://Python4Capella/simplified_api/capella.
    py')
8 if False:
9     from simplified_api.capella import *
10
11 # include to read/write xlsx files
12 from openpyxl import *
13

```

```

14 # retrieve the element from the current selection
15 selected_elem = CapellaElement(CapellaPlatform.
    getFirstSelectedElement())
16
17 # insert the path of your Capella model
18 aird_path = ''
19
20 model = CapellaModel()
21 model.open(aird_path)
22
23 hierarchy_data = []
24
25 # go through the hierarchy of Logical Components and places
    it on hierarchy_data
26 def traverseHierarchy(elem, parent_name=None, level=1):
27     elem_name = elem.get_name()
28     hierarchy_data.append({
29         'Component Name': elem_name,
30         'Parent Name': parent_name,
31         'Hierarchy Level': level
32     })
33
34     # if the current element is a Logical Component, search
    for children
35     if isinstance(elem, LogicalComponent):
36         lc = elem
37         for sub_elem in lc.get_owned_logical_components():
38             if isinstance(sub_elem, LogicalComponent):
39                 # recursive function, passing the name of
    the current element as parent
40                 traverseHierarchy(sub_elem, elem_name, level
    + 1)
41
42 # get System Engineering
43 se = model.get_system_engineering()
44 print('starting export of model ' + se.get_name())
45
46 # prepare Excel file export
47 project_name = aird_path[0:(aird_path.index("/", 1) + 1)]
48 project = CapellaPlatform.getProject(project_name)
49 folder = CapellaPlatform.getFolder(project, 'results')
50 xlsx_file_name = CapellaPlatform.getAbsolutePath(folder) + '
    /' + 'Export_LC.xlsx'
51
52 # create a workbook

```

```

53 workbook = Workbook()
54
55 # write Excel file header
56 worksheet = workbook.active
57 worksheet.title = 'LC export'
58 headers = ['Component Name', 'Parent Name', 'Hierarchy Level']
59 worksheet.append(headers)
60
61 # search for first level Logical Components and call the
    function
62 for elem in se.get_logical_architecture().get_logical_system
    ().get_owned_logical_components():
63     if isinstance(elem, LogicalComponent):
64         traverseHierarchy(elem)
65
66 # write the list on Excel
67 for row_data in hierarchy_data:
68     worksheet.append([
69         row_data['Component Name'],
70         row_data['Parent Name'],
71         row_data['Hierarchy Level']
72     ])
73
74 # save the Excel file
75 workbook.save(xlsx_file_name)
76
77 # refresh
78 CapellaPlatform.refresh(folder)

```

Listing A.3: *Export\_Logical\_Component.py* script [22].

## A.4 *Export\_modes.py*

```

1 # include for the Capella modeller API
2 include('workspace://Python4Capella/simplified_api/capella.
    py')
3 if False:
4     from simplified_api.capella import *
5
6 # include to read/write xlsx files
7 from openpyxl import *
8
9 # insert the path of your Capella model

```

```

10 aird_path = ''
11
12 model = CapellaModel()
13 model.open(aird_path)
14
15 # examine the Logical Architecture level
16 se = model.get_system_engineering()
17 la = se.get_logical_architecture()
18 log_comp = la.get_logical_component_pkg()
19
20 # get all modes
21 modes = log_comp.get_all_contents_by_type(Mode)
22
23 # prepare Excel file export
24 project_name = aird_path[0:(aird_path.index("/", 1) + 1)]
25 project = CapellaPlatform.getProject(project_name)
26 folder = CapellaPlatform.getFolder(project, 'script results'
27 )
28
29 xlsx_file_name = CapellaPlatform.getAbsolutePath(folder) + '
30 /' + 'Export_modes.xlsx'
31
32 # create a workbook
33 workbook = Workbook()
34
35 # write Excel file header
36 worksheet = workbook.active
37 worksheet.title = 'Operative modes'
38 worksheet["A1"] = 'Modes'
39
40 i = 2
41
42 for mode in modes:
43     worksheet["A" + str(i)] = mode.get_label()
44     i = i + 1
45
46 # save the Excel file
47 workbook.save(xlsx_file_name)
48
49 # refresh
50 CapellaPlatform.refresh(folder)

```

Listing A.4: *Export\_modes.py* script [22].

## Appendix B

# Valispace Python API scripts

### B.1 *Import\_Requirements.py*

```
1 import valispace
2 import pandas as pd
3 from tqdm import tqdm
4
5 deployment = input("Deployment Name:")
6
7 # Authentication to Valispace
8 try:
9     valispace = valispace.API(url="https://" + deployment + ".
10     valispace.com",
11     session_token="Bearer ...") # insert your access token
12     print("Successful authentication using the provided
13     Token")
14
15 except Exception as e:
16     print(f"An error has occurred: {e}")
17
18 # insert your Excel file path
19 csvFilePath = r"..."
20
21 # insert the ID of the specification the requirements should
22 # added to
23 specification_ID = int(input("Enter the ID of the project:"))
```



```

21
22 # generate the new identifier for the imported requirement
23 def getNextReqIdentifier(last_req_identifier, increase = 10)
24 :
25     splitted_req_identifier = last_req_identifier.split('
26     -')
27     splitted_req_identifier_len = len(
28     splitted_req_identifier)
29     next_index = int(splitted_req_identifier[
30     splitted_req_identifier_len-1]) + increase
31     next_idenfier = f"{'-'}.join(splitted_req_identifier[:
32     splitted_req_identifier_len-1]))-{f"{'str(next_index):03}"
33     }"
34     return next_idenfier
35
36 # get the identifier of the last requirement created in that
37     specification
38 def getLastReqIdentifier(specificationID):
39     try:
40         SpecRequirements = valispace.get(f'
41         requirements/specifications/{specificationID}/')
42         SpecRequirements = SpecRequirements['
43         requirements']
44         req_identifiers = []
45         for req in tqdm(SpecRequirements, desc="
46         Getting informations of the specification requirements"):
47             req_identifier = valispace.get(f'
48             requirements/{req}')['identifier']
49             req_identifiers.append(req_identifier
50         )
51
52         last_identifier = ""
53         biggest_index = 0
54         for identifier in req_identifiers:
55             splitted_identifier = identifier.
56             split('-')
57             index = int(splitted_identifier[len(
58             splitted_identifier)-1])
59             if index > biggest_index:
60                 biggest_index = index
61                 last_identifier = identifier
62         return last_identifier
63     except:
64         Exception('Error while getting specification
65         requirements!')
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

51 # get all requirements in the selected specification
52 def import_req(csvFilePath, specification_ID):
53     reqs = pd.read_excel(csvFilePath)
54     last_req_identifier = getLastReqIdentifier(
55         specification_ID)
56     for index, row in reqs.iterrows():
57         next_req_identifier = getNextReqIdentifier(
58             last_req_identifier)
59         last_req_identifier = next_req_identifier
60         req = {
61             "specification": specification_ID,
62             "identifier" : next_req_identifier,
63             "title" : row['Function'],
64             "text" : row['Description']
65         }
66         requirementPosted = valispace.post('
requirements/', req)
import_req(csvFilePath, specification_ID)

```

Listing B.1: *Import\_Requirements.py* script.

## B.2 *Import\_System\_Components.py*

```

1 import valispace
2 import pandas as pd
3
4 deployment = input("Deployment Name:")
5
6 # Authentication to Valispace
7 try:
8     valispace = valispace.API(url="https://" + deployment + ".
9     valispace.com",
10     session_token="Bearer ...") # insert your access token
11     print("Successful authentication using the provided
12     Token")
13
14 except Exception as e:
15     print(f"An error has occurred: {e}")
16
17 # insert the Excel file path
18 excelFilePath = r"..."
19
20 # insert the ID of the parent component

```

```

19 parent_component = 0
20
21 # import System Components
22 def import_comp(excelFilePath, parent_component):
23     comps = pd.read_excel(excelFilePath)
24     for index, row in comps.iterrows():
25         component = {
26             "name": row['Name'],
27             "parent": parent_component
28         }
29         # replace spaces with underscores
30         for key, value in component.items():
31             if isinstance(value, str):
32                 component[key] = value.replace(' ', '_')
33         componentPosted = valispace.post("components/",
34                                         component)
35
36 import_comp(excelFilePath, parent_component)

```

Listing B.2: *Import\_System\_Components.py* script.

### B.3 *Import\_LC\_Hierarchy.py*

```

1 import valispace
2 import pandas as pd
3
4 deployment = input("Deployment Name:")
5
6 # Authentication to Valispace
7 try:
8     valispace = valispace.API(url="https://" + deployment + ".valispace.com",
9                               session_token="Bearer ...") # insert your access token
10    print("Successful authentication using the provided Token")
11
12 except Exception as e:
13     print(f"An error has occurred: {e}")
14
15 capella_id_to_valispace_id_map = {}
16
17 # insert the Excel file path
18 excelFilePath = r"..."
19 name_to_valispace_id_map = {}

```

```

20
21 # inserts the ID of the parent component from which to start
   the import
22 top_parent_id_in_valispace = 0
23
24 # hierarchy import
25 def import_full_hierarchy_by_name(excelFilePath,
   top_level_valispace_parent_id=0):
26     print(f"Start importing LC hierarchy from file: {
   excelFilePath}")
27     try:
28         comps_df = pd.read_excel(excelFilePath)
29         all_levels = sorted(comps_df['Hierarchy Level'].
   unique())
30         for current_level in all_levels:
31             components_at_current_level = comps_df[comps_df[
   'Hierarchy Level'] == current_level]
32
33             # replace spaces with underscores
34             for index, row in components_at_current_level.
   iterrows():
35                 current_elem_name = row['Component Name']
36                 parent_elem_name_from_excel = row['Parent
   Name']
37                 current_elem_name_cleaned =
   current_elem_name.replace(' ', '_').replace('&', '_')
38                 if pd.isna(parent_elem_name_from_excel):
39                     parent_elem_name_cleaned =
   parent_elem_name_from_excel.replace(' ', '_').replace('&',
   '_')
40                 else:
41                     parent_elem_name_cleaned = None
42                 # determine the ID of the parent of the
   current component
43                 valispace_parent_id = None
44                 if current_level == 1:
45                     valispace_parent_id =
   top_level_valispace_parent_id
46                 else:
47                     valispace_parent_id =
   name_to_valispace_id_map.get(parent_elem_name_cleaned)
48                     if not valispace_parent_id:
49                         print(f" Attention: parent ID '{
   parent_elem_name_from_excel}' for '{current_elem_name}'
   not found.")

```

```

50         continue
51
52         component_data = {
53             "name": current_elem_name_cleaned,
54         }
55         if valispace_parent_id:
56             component_data["parent"] =
valispace_parent_id
57
58         try:
59             # import the component on Valispce
60             component_posted = valispace.post("
components/", component_data)
61             valispace_id = component_posted.get('id'
)
62             valispace_name_from_vs =
component_posted.get('name')
63             name_to_valispace_id_map[
current_elem_name_cleaned] = valispace_id
64
65             except Exception as e:
66                 print(f" Error: {e}")
67                 continue
68
69             print("\nImport of hierarchy completed.")
70
71         except FileNotFoundError:
72             print(f"Error: '{excelFilePath}' excel file not found
")
73         except Exception as e:
74             print(f"Error: {e}")
75
76 import_full_hierarchy_by_name(excelFilePath,
top_parent_id_in_valispace)

```

Listing B.3: *Import\_LC\_Hierarchy.py* script.

## B.4 *Components\_bridge.py*

```

1 import valispace
2 import pandas as pd, json
3
4 deployment = input("Deployment Name:")
5

```

```

6  # Authentication to Valispace
7  try:
8      valispace = valispace.API(url="https://" + deployment + ".
          valispace.com",
9      session_token="Bearer ...")    # insert your access token
10     print("Successful authentication to Valispace using the
          provided Token")
11
12 except Exception as e:
13     print(f"An error has occurred: {e}")
14
15 # insert the Valispace project ID
16 Valispace_project_ID = 100
17
18 # specify the ID of the component from which to start the
          hierarchy
19 start_component_ID = 30856
20
21 # Excel file of the Logical Component
22 excel_file_path = r"..." exported from Capella
23 excel_sheet_name = 'LC export'
24 excel_parent_column = 'Parent Name'
25 excel_component_column = 'Component Name'
26
27 # Function that extracts all components from Valispace
          project ID
28 def get_all_project_components(project_id):
29     all_components = valispace.get(f"components/?project={
          str(project_id)}")
30     return all_components
31
32 # Function that builds the parent-child hierarchy of
          components
33 def get_valispace_parent_child_hierarchy(all_components,
          start_id=None):
34
35     component_id_to_name = {comp['id']: comp['name'] for
          comp in all_components}
36     component_id_to_obj = {comp['id']: comp for comp in
          all_components}
37
38     # Search for the starting component of the hierarchy
39     first_component = None
40     if start_id:
41         first_component = component_id_to_obj.get(start_id)

```

```

42         if not first_component:
43             print(f"WARNING: starting component not found")
44             return []
45     else:
46         print("ERROR: Define the correct ID")
47         return []
48
49     # Recursive internal function to extract the parent-
child relationships
50     def get_relationships_recursive(current_component_id):
51         Valispace_relationships = []
52         current_component_name = component_id_to_name.get(
current_component_id)
53
54         # Find all direct children of the current component
55         children = []
56         for comp in all_components:
57             if comp.get('parent') == current_component_id:
58                 children.append(comp)
59
60         for child in children:
61             child_name = child['name']
62             Valispace_relationships.append({"Parent":
current_component_name, "Child": child_name})
63
64         # Call the function recursively for the child's
children
65         Valispace_relationships.extend(
get_relationships_recursive(child['id']))
66         return Valispace_relationships
67
68         # Start recursion from the chosen initial component
69         parent_child_relationships = get_relationships_recursive
(first_component['id'])
70         return parent_child_relationships
71
72     # Function that reads Excel data from Capella
73     def get_excel_data(file_path, sheet_name, parent_col,
child_col):
74         excel = pd.read_excel(file_path, sheet_name=sheet_name)
75
76         Capella_relationships = []
77
78         for index, row in excel.iterrows():
79             # Retrieves values and converts them into strings

```

```

80     parent = str(row[parent_col]).strip() if pd.notna(
row[parent_col]) else None
81     child = str(row[child_col]).strip() if pd.notna(row[
child_col]) else None
82
83     # Replace spaces with underscores
84     if parent is not None:
85         parent = parent.replace(" ", "_").replace("&", "
_")
86     if child is not None:
87         child = child.replace(" ", "_").replace("&", "_
")
88
89     # Normalise the string 'None' or empty to a true
None for the parent
90     if parent == '' or (isinstance(parent, str) and
parent.lower() == 'none'):
91         parent = None
92
93     if child:
94         Capella_relationships.append({"Parent": parent,
"Child": child})
95
96     return Capella_relationships
97
98
99 # Function that compares the component relationships of
Capella and Valispace
100 def compare_lists(valispace_relationships,
capella_relationships):
101     valispace_rel_set = set((rel['Parent'] if rel['Parent']
is not None else '', rel['Child']) for rel in
valispace_relationships)
102     capella_rel_set = set((rel['Parent'] if rel['Parent'] is
not None else '', rel['Child']) for rel in
capella_relationships)
103
104     rel_only_in_valispace = valispace_rel_set -
capella_rel_set
105     rel_only_in_capella = capella_rel_set -
valispace_rel_set
106
107     if not rel_only_in_valispace and not rel_only_in_capella
:

```



```

108         print("Parent-Child relationships are IDENTICAL in
Valispace and Capella")
109     else:
110         if rel_only_in_valispace:
111             print("\nRelationships PRESENT in Valispace but
NOT in Capella:")
112             for parent, child in sorted(list(
rel_only_in_valispace)):
113                 print(f"    - Parent: '{parent if parent else
'None'}', Child: '{child}'")
114
115         if rel_only_in_capella:
116             print("\nRelationships PRESENT in Capella but
NOT in Valispace:")
117             for parent, child in sorted(list(
rel_only_in_capella)):
118                 print(f"    - Parent: '{parent if parent else
'None'}', Child: '{child}'")
119
120 # Function that defines the components to be imported into
Valispace
121 def comps_to_import_to_Vali(valispace_relationships,
capella_relationships):
122     capella_rel_set = set((rel['Parent'] if rel['Parent'] is
not None else '', rel['Child']) for rel in
capella_relationships)
123     valispace_comps_set = set((rel['Child']) for rel in
valispace_relationships)
124     capella_comps_set = set((rel['Child']) for rel in
capella_relationships)
125     comps_only_in_capella = capella_comps_set -
valispace_comps_set
126     only_capella_child_parent = {child: parent for parent,
child in capella_rel_set}
127
128     components_to_import_to_valispace = []
129     for comp in comps_only_in_capella:
130         parent_of_comp = only_capella_child_parent.get(comp,
'',)
131         components_to_import_to_valispace.append({
132             "name": comp,
133             "parent": parent_of_comp
134         })
135     if not comps_only_in_capella:

```

```
136         print("\nValispace and Capella have the same
components")
137     else:
138         print("\nComponents that are PRESENT in Capella but
NOT in Valispace:")
139         print(f"\n{components_to_import_to_valispace}")
140
141     print("\nComparison completed.")
142     return components_to_import_to_valispace
143
144
145 print("Start of Valispace and Capella data extraction
programme...")
146
147 all_project_components = get_all_project_components(
    Valispace_project_ID)
148
149 valispace_hierarchy = []
150 if all_project_components:
151     valispace_hierarchy =
152     get_valispace_parent_child_hierarchy(
153         all_project_components,
154         start_id=start_component_ID
155     )
156
157 capella_data = get_excel_data(
158     excel_file_path,
159     excel_sheet_name,
160     excel_parent_column,
161     excel_component_column
162 )
163
164 if valispace_hierarchy and capella_data:
165     compare_lists(valispace_hierarchy, capella_data)
166 else:
167     print("\nUnable to compare: no valid data from Valispace
or Capella.")
168
169 new_vali_comps = comps_to_import_to_Vali(valispace_hierarchy
, capella_data)
170
171 if new_vali_comps:
172     print("\n--- Import to Valispace ---")
173
174     for component_data in new_vali_comps:
```

```

174     component_name = component_data['name']
175     parent_name = component_data['parent']
176     parent_id = None # initialisation
177
178     if parent_name:
179         try:
180             search_parent_ID = valispace.get('components
181 ', params={
182                 'name': parent_name,
183                 'project': Valispace_project_ID
184             })
185             if search_parent_ID:
186                 for found_component in search_parent_ID:
187                     if found_component.get('name') ==
parent_name:
188                         parent_id = found_component.get(
'id')
189                         print(f"SUCCESS: Parent '{
parent_name}' with {parent_id} was found.")
190                         break
191
192                     if parent_id is None:
193                         print(f"WARNING: Parent '{
parent_name}' of the component '{component_name}' not
found. The component will be created without a parent.")
194                     else:
195                         print(f"WARNING: No results found for
the parent '{parent_name}'. The component '{
component_name}' was not imported.")
196                         continue
197
198             except Exception as e:
199                 print(f"ERROR: {e} \nParent's ID search
failed for the '{component_name}'.")
200                 continue
201
202     valispace_payload = {
203         "name": component_name,
204         "project": Valispace_project_ID
205     }
206     if parent_id is not None:
207         valispace_payload["parent"] = parent_id
208
209     # Post the component on Valispace

```

```

210         component_posted = valispace.post("components/",
211                                           valispace_payload)
212     else:
213         print("\nThere are no components to import.")

```

Listing B.4: *Components\_bridge.py* script.

## B.5 *Import\_All\_Modelists.py*

```

1  import valispace
2  import pandas as pd
3
4  deployment = input("Deployment Name:")
5
6  # Authentication to Valispace
7  try:
8      valispace = valispace.API(url="https://" + deployment + ".
9      valispace.com",
10      session_token="Bearer ...") # insert your access token
11      print("Successful authentication using the provided
12      Token")
13
14  except Exception as e:
15      print(f"An error has occurred: {e}")
16
17  # insert the ID of the project
18  project_ID = 0
19  # insert the ID of the component of interest
20  component_ID = 0
21  # insert the name of new Modelist to be imported
22  modelist_name = "Operative Modes"
23
24  # insert Excel file path
25  excel_file_path = r"..."
26
27  # Import Modelist on Valispace
28  def import_modes(excelFilePath, component_ID, modelist_name)
29  :
30      modes = pd.read_excel(excelFilePath)
31      mode_name_string = [
32          [str(row['Modes']).strip().replace(' ', '_')]
33          for index, row in modes.iterrows()
34          if str(row['Modes']).strip()

```

```
32     ]
33     num_rows = len(mode_name_string)
34     cleaned_modelist = modelist_name.strip().replace(' ', '_')
35     modelist = {
36         "mode_names": mode_name_string,
37         "parent": component_ID,
38         "name": cleaned_modelist,
39         "number_of_rows": num_rows,
40         "matrices_referring": [],
41         "matrices_linking": []
42     }
43     print(modelist)
44     modePosted = valispace.post("modelists/", modelist)
45
46 import_modes(excel_file_path, component_ID, modelist_name)
47 print("Modelist was imported")
```

**Listing B.5:** *Import\_All\_Modelists.py* script.

# Appendix C

## Capella diagrams

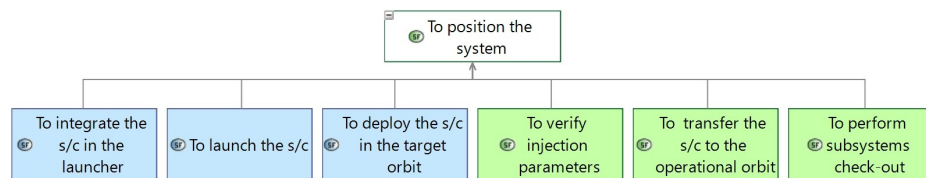


Figure C.1: SFBD To position the system

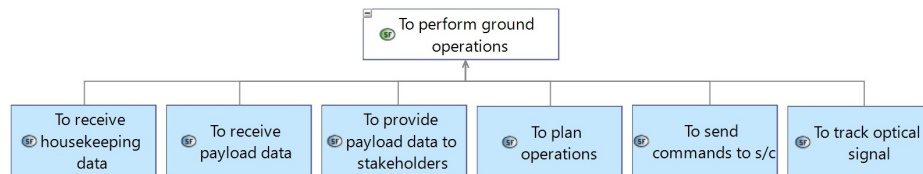


Figure C.2: SFBD To perform ground operations

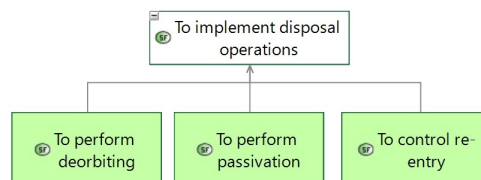


Figure C.3: SFBD To implement disposal operations

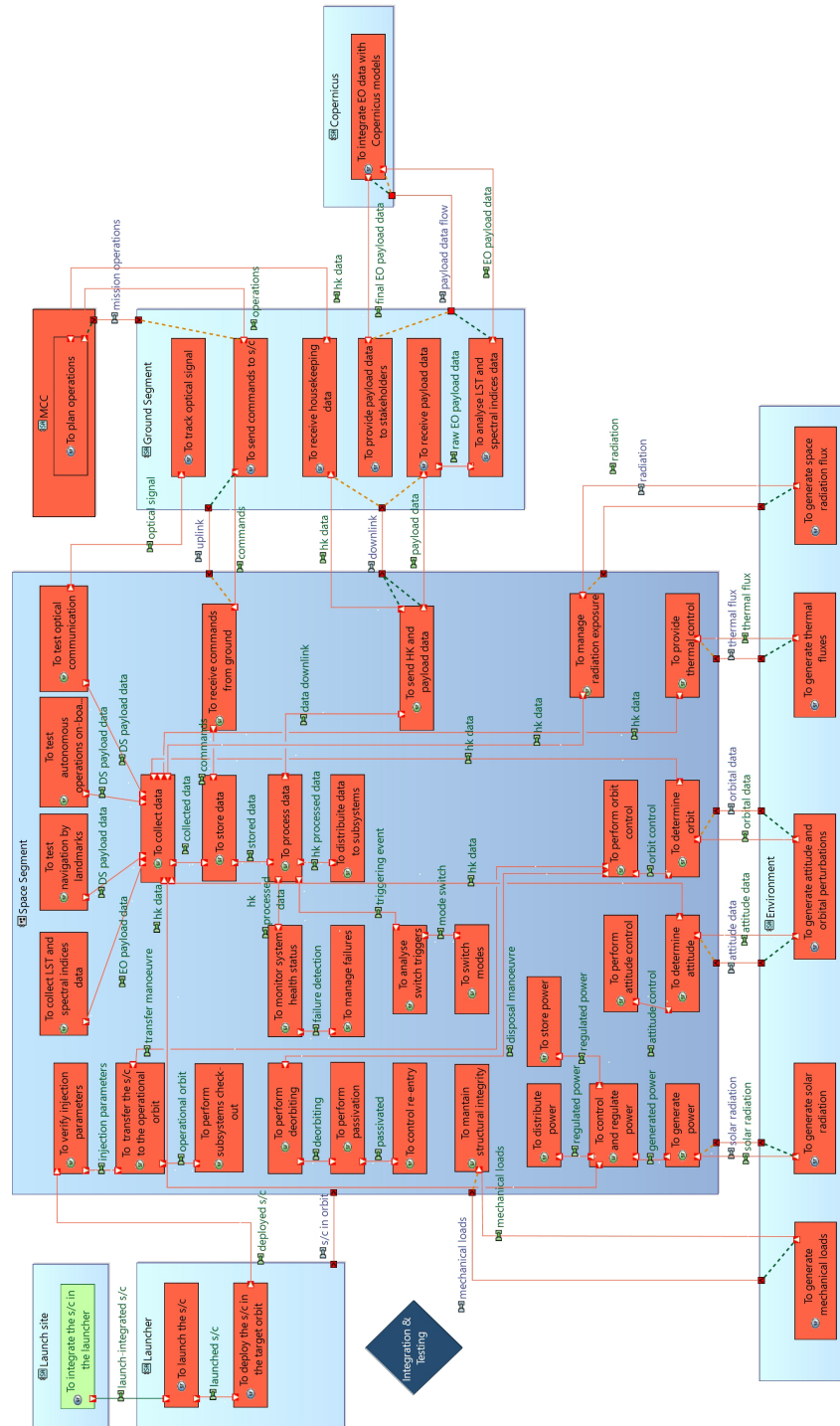
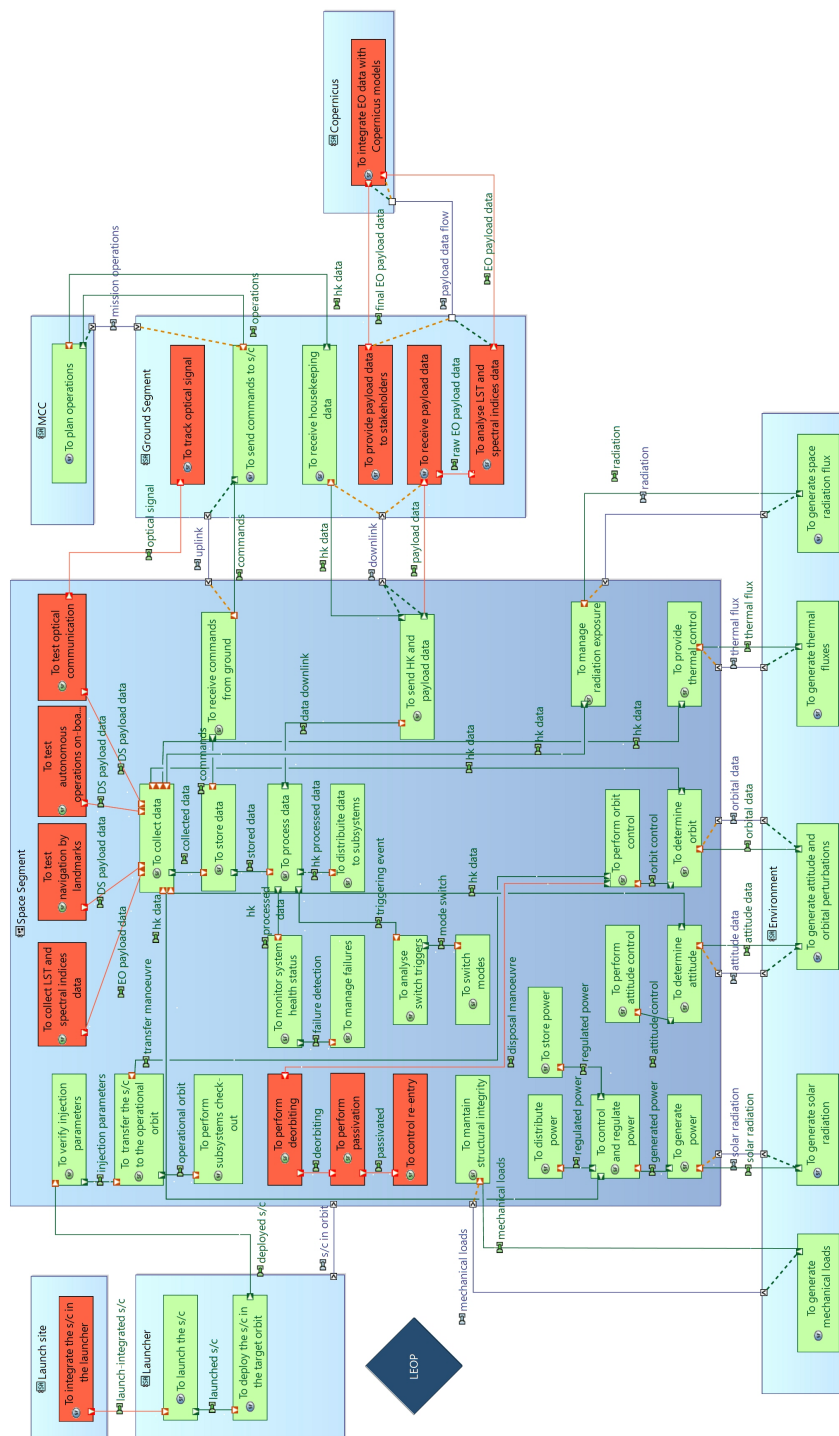


Figure C.4: SAB I&T



**Figure C.5:** SAB LEOP



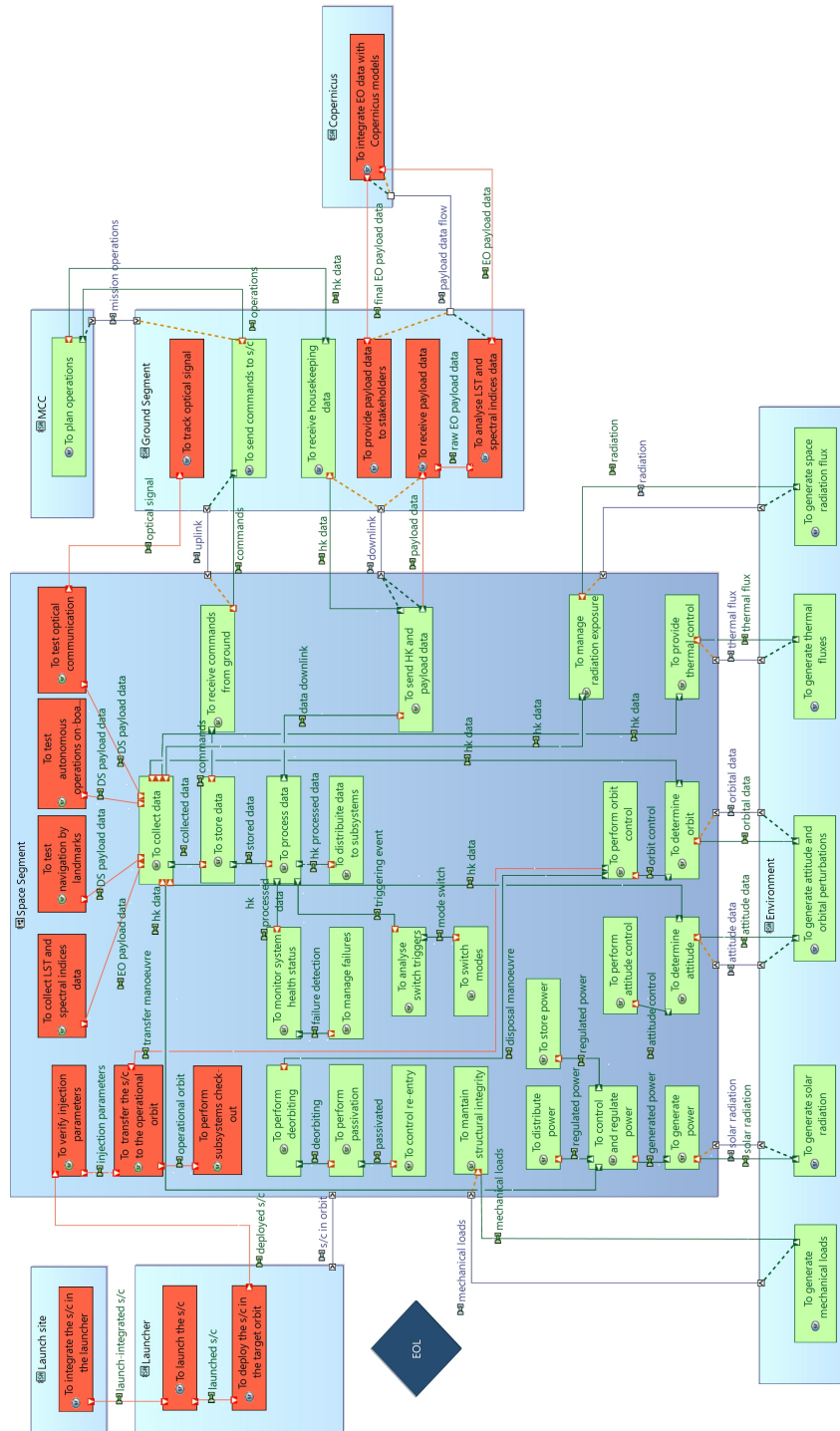


Figure C.6: SAB EOL

Component	Mass	margin +	total margin +	worst case +	margin -	total margin -	worst case -
▼ Space_Segment	27.74 kg	+20%	+26%	34.952 kg	-0%	0%	27.74 kg
▼ Payload	5.12 kg	+0%	+5%	5.376 kg	-0%	0%	5.12 kg
▼ Deep_Space	2.16 kg	+0%	+5%	2.268 kg	-0%	0%	2.16 kg
▼ Navigation	1.5 kg	+0%	+5%	1.575 kg	-0%	0%	1.5 kg
— VNIR_Camera	1.5 kg	+5%	+5%	1.575 kg	-0%	0%	1.5 kg
▼ Optical_communication	0.66 kg	+0%	+5%	0.693 kg	-0%	0%	0.66 kg
— Laser_Module	0.66 kg	+5%	+5%	0.693 kg	-0%	0%	0.66 kg
▼ EO	2.96 kg	+0%	+5%	3.108 kg	-0%	0%	2.96 kg
— LWIR_Camera	1.8 kg	+5%	+5%	1.89 kg	-0%	0%	1.8 kg
— SWIR_Camera	1.16 kg	+5%	+5%	1.218 kg	-0%	0%	1.16 kg
▼ Platform	22.62 kg	+0%	+5%	23.751 kg	-0%	0%	22.62 kg
> AODS__GNC	5.39 kg	+0%	+5%	5.659 kg	-0%	0%	5.39 kg
— C_DH	0.28 kg	+5%	+5%	0.294 kg	-0%	0%	0.28 kg
— Electronics	0.3 kg	+5%	+5%	0.315 kg	-0%	0%	0.3 kg
> EPS	4.43 kg	+5%	+5%	4.651 kg	-0%	0%	4.43 kg
— STR__MECH	10.5 kg	+5%	+5%	11.025 kg	-0%	0%	10.5 kg
— TCS	0.24 kg	+5%	+5%	0.252 kg	-0%	0%	0.24 kg
— TT_C	1.48 kg	+5%	+5%	1.554 kg	-0%	0%	1.48 kg
Total	27.74 kg	+20%	+26%	34.952 kg	-0%	0%	27.74 kg

Figure C.7: Mass budget estimation on Valispace.

Component	Volume	margin +	total margin +	worst case +	margin -	total margin -	worst case -
▼ Space_Segment	19.95 dm³	+20%	+26%	25.137 dm³	-0%	0%	19.95 dm³
▼ Payload	5 dm³	+0%	+5%	5.25 dm³	-0%	0%	5 dm³
▼ Deep_Space	2 dm³	+0%	+5%	2.1 dm³	-0%	0%	2 dm³
▼ Navigation	1.5 dm³	+0%	+5%	1.575 dm³	-0%	0%	1.5 dm³
— VNIR_Camera	1.5 dm³	+5%	+5%	1.575 dm³	-0%	0%	1.5 dm³
▼ Optical_communication	0.5 dm³	+0%	+5%	0.525 dm³	-0%	0%	0.5 dm³
— Laser_Module	0.5 dm³	+5%	+5%	0.525 dm³	-0%	0%	0.5 dm³
▼ EO	3 dm³	+0%	+5%	3.15 dm³	-0%	0%	3 dm³
— LWIR_Camera	1.44 dm³	+5%	+5%	1.512 dm³	-0%	0%	1.44 dm³
— SWIR_Camera	1.56 dm³	+5%	+5%	1.638 dm³	-0%	0%	1.56 dm³
▼ Platform	14.95 dm³	+0%	+5%	15.698 dm³	-0%	0%	14.95 dm³
> AODS__GNC	8.4 dm³	+0%	+5%	8.82 dm³	-0%	0%	8.4 dm³
— C_DH	0.42 dm³	+5%	+5%	0.441 dm³	-0%	0%	0.42 dm³
— Electronics	0.6 dm³	+5%	+5%	0.63 dm³	-0%	0%	0.6 dm³
> EPS	2.55 dm³	+5%	+5%	2.677 dm³	-0%	0%	2.55 dm³
— STR__MECH	1.67 dm³	+5%	+5%	1.754 dm³	-0%	0%	1.67 dm³
— TCS	0.21 dm³	+5%	+5%	0.221 dm³	-0%	0%	0.21 dm³
— TT_C	1.1 dm³	+5%	+5%	1.155 dm³	-0%	0%	1.1 dm³
Total	19.95 dm³	+20%	+26%	25.137 dm³	-0%	0%	19.95 dm³

Figure C.8: Volume budget estimation on Valispace.

# Bibliography

- [1] NASA. *NASA System Engineering Handbook*. NASA, 2007 (cit. on pp. 3, 4).
- [2] D.D. Walden, G.J. Roedler, K.J. Forsberg, R.D. Hamelin, and T.M. Shortell. *INCOSE Systems engineering handbook*. Wiley, 2015 (cit. on pp. 3, 8).
- [3] S. Corpino. *Space Missions and Systems Design – SMSD*. Tech. rep. lectures, Politecnico di Torino, 2025/26 (cit. on pp. 3, 11).
- [4] E. Brusa, A. Calà, and D. Ferretto. *System Engineering and Its Application to Industrial Product Development*. Springer, 2018 (cit. on pp. 4, 11–15).
- [5] L. Lindblad, M. Witzmann, and S. Vanden Bussche. *Data-Driven System Engineering: turning MBSE into industrial reality*. 2021. URL: <https://www.valispace.com/wp-content/uploads/2021/03/Valispace-2018-Turning-MBSE-Into-Industrial-Reality-SECESA.pdf> (cit. on pp. 5–8).
- [6] M. Ackley. *Efficient Engineering 101 (II): Document-Driven vs. Data-Driven Systems Engineering (DDSE)*. 2019. URL: <https://www.valispace.com/efficient-engineering-101-ii-document-driven-vs-data-driven-systems-engineering-ddse/> (cit. on pp. 5, 17).
- [7] Jeff A. Estefan. «Survey of Model-Based Systems Engineering (MBSE) Methodologies». In: *ResearchGate.net* (2008) (cit. on pp. 8, 10, 11).
- [8] NDIA. *Model-Based Engineering Subcommittee, Final Report*. Tech. rep. NDIA, 2011 (cit. on p. 8).
- [9] M. Peres. *The Complete Guide to Model-Based Systems Engineering (MBSE)*. Valispace. 2023. URL: <https://www.valispace.com/the-complete-guide-to-model-based-systems-engineering-mbse/> (cit. on p. 8).
- [10] P. Guardabasso, L. Lindbland, M. Witzmann, and S. Siarov. «Evaluation of the Learning Process of a Data-Driven Systems Engineering Methodology in a Workshop Environment». In: *International Astronautical Congress (IAC-19)*. 2019 (cit. on p. 9).

- [11] G. Dinolfo. «Application of MBSE to reverse engineering a rendezvous and docking space mission». MA thesis. Politecnico di Torino, 2022 (cit. on p. 9).
- [12] IBM. *Integrating Rhapsody and DOORS*. 2025. URL: <https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/10.0.1?topic=tools-integrating-rhapsody-doors> (cit. on p. 15).
- [13] Dassault Systemes. *Cameo System Modeler*. 2025. URL: <https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler> (cit. on p. 15).
- [14] MathWorks. *System Composer*. 2025. URL: <https://www.mathworks.com/products/system-composer.html> (cit. on p. 16).
- [15] Eclipse. *Eclipse Papyrus*. 2023. URL: <https://eclipse.dev/papyrus/> (cit. on p. 16).
- [16] L. Lindblad, M. Witzmann, and S. Vanden Bussche. *System Engineering fro a web browser: turning MBSE into industrial reality*. 2016. URL: <https://www.valispace.com/wp-content/uploads/2021/02/Lindblad-SECESA-2016-Valispace-web-browser-engineering.pdf> (cit. on pp. 16–18).
- [17] Capella. URL: <https://mbse-capella.org/arcadia.html> (cit. on pp. 16, 20, 21, 23, 24).
- [18] URL: <https://www.valispace.com/> (cit. on p. 18).
- [19] P. Roques. *System Architecture Modeling with the Arcadia Method - A pratical guide to Capella*. ISTE, 2018 (cit. on pp. 20–23, 42, 49, 51, 59, 65, 69).
- [20] P. Minacapilli, F. C. Zurita, S. Criado Zurita, S. Campo Pérez, A. Rodríguez Pérez-Silva, and D. Escudero Lasheras. «Small satellites mission design enhancement through MBSE and DDSE toolchain». In: *Model Based Space Systems and Software Engineering - MBSE2022* (2022) (cit. on pp. 28, 29, 62).
- [21] J. Whitehouse. *MBSE at ESA: State of MBSE in ESA Missions and Activities*. MBSE 2021 Conference. URL: <https://indico.esa.int/event/386/timetable/#5-mbse-at-esa-state-of-mbse-in> (cit. on p. 29).
- [22] Labs for Capella. *Python for Capella*. Valispace. URL: <https://github.com/labs4capella/python4capella> (cit. on pp. 30, 34, 35, 78, 79, 81, 82).
- [23] *Valispace Python API*. URL: <https://github.com/valispace/ValispacePythonAPI> (cit. on p. 30).
- [24] Capella. *Capella Add-Ons*. URL: <https://mbse-capella.org/addons.html> (cit. on p. 33).

- [25] M2Doc from Capella model. *Python4Capella Simplified Metamodel*. URL: <https://github.com/labs4capella/python4capella/blob/master/specification/M2Doc%20generation/Python4Capella%20Simplified%20Metamodel.docx> (cit. on p. 35).
- [26] ECMWF Copernicus Climate Change Service. *Demonstrating heat stress in European cities*. 2019. URL: <https://climate.copernicus.eu/demonstrating-heat-stress-european-cities> (cit. on pp. 41, 43).
- [27] Copernicus Land Monitoring Service. *A method to combat the Urban Heat Island effect*. URL: <https://land.copernicus.eu/en/use-cases/a-method-to-combat-the-urban-heat-island-effect/a-method-to-combat-the-urban-heat-island-effect> (cit. on p. 41).
- [28] Copernicus Climate Data Store. *Climate variables for cities in Europe from 2008 to 2017*. 2019. URL: <https://cds.climate.copernicus.eu/datasets/sis-urban-climate-cities?tab=overview> (cit. on p. 43).
- [29] ECMWF. *Destination Earth Use Case. Addressing urban heat island effect*. URL: <https://stories.ecmwf.int/destination-earth-use-case/> (cit. on p. 43).
- [30] Cure Copernicus. *Copernicus for Urban Resilience in Europe*. URL: <https://cure-copernicus.eu/thecuresystem.html> (cit. on p. 43).
- [31] ESA. *Urban Heat Islands and Urban Thermography*. Tech. rep. The UHI project - Executive Summary, 2011 (cit. on p. 43).
- [32] AAC Clyde Space. *OPTIMUS-80*. Datasheet. URL: <https://www.aac-clyde.space/what-we-do/space-products-components/cubesat-batteries> (cit. on p. 71).
- [33] GOMspace. *NanoPower BPX*. Datasheet. URL: <https://gomspace.com/shop/subsystems/power/nanopower-bpx.aspx> (cit. on p. 71).
- [34] Github. *Python-Capellambse*. URL: <https://github.com/DSD-DBS/py-capellambse> (cit. on p. 73).