Politecnico di Torino

Politecnico di Torino

Master's Degree in Aerospace Engineering

A.Y. 2025/2026

Graduation Session April 2026

Model-Based System Engineering for Space Missions & Systems Design

Examining System Composer's capabilities through CubeSat and ISS ECLSS Applications

Academic Supervisors:

Prof. Sabrina Corpino
Ing. Serena Campioli

Candidate:
Federico Quaglia

Abstract

The increasing complexity of modern space systems and stricter performance and safety requirements have highlighted the limitations of traditional document-based systems engineering, particularly in early design phases, motivating the adoption of Model-Based Systems Engineering (MBSE) as a model-centric approach.

This thesis investigates the potential of MBSE for space system conceptual and early-stage design, in particular examining *System Composer* comparing it with other MBSE tools to assess its capability to support the standardisation and scalability of system architectures across multiple levels of abstraction, and demonstrate its effectiveness through realistic space engineering case studies.

A comprehensive literature review of MBSE practices and tools has been performed together with practical modelling activities implemented in System Composer and Simulink. The first case study focuses on the conceptual design of a CubeSat mission, illustrating the application of MBSE principles at both mission and system levels. Once the full potential has been disclosed, the human-factor as the central payload has been integrated through Human-Centred Engineering principles to apply MBSE to a second representative case study addressing the Environmental Control and Life Support System (ECLSS) of the International Space Station.

The results show that System Composer provides an accessible and flexible MBSE environment, particularly suited to educational and research contexts due to its integration within the MATLAB/Simulink ecosystem. Its block-based architecture and built-in requirements management capabilities support coherent architectural modelling, traceability, and early verification activities. Overall, the thesis demonstrates that System Composer represents a valuable compromise between usability and modelling power for conceptual space system design, while also highlighting directions for future developments and applications within the MBSE framework.

Keywords: Model-Based Systems Engineering, System Composer, Simulink, Space Systems Engineering, CubeSat, Human-Centred Engineering

Table of Contents

List of Figures

List of Tables

List of the abbreviations

AAA	Avionics Air Assembly
ACS	Atmosphere Control and Supply
AL	Air Lock
AR	Atmosphere Revitalisation
CCAA	Common Cabin Air Assembly
CDRA	The Carbon Dioxide Removal Assembly
CHeC	Crew Health Care
CMS	Countermeasures System
ConOps	Concept of Operations
DBSE	Document-Based System Engineering
DDSE	Data-Driven System Engineering
ECSS	European Coordination for Space Standardization
EHS	Environmental Health Subsystem
ERA	European Robotic Arm
ESA	European Space Agency
EVA	Extra-Vehicular Activity
FDS	Fire Detection and Suppression
GRCs	Galactic Cosmic Rays
HMS	Health Maintenance System
IMV	Intermodule Ventilation
INCOSE	International Council on Systems Engineering
ITCS	Internal Thermal Control System
JPL	Jet Propulsion Laboratory
MBSE	Model Based System Engineering
MCA	Major Constituents Analyser
OGA	Oxygen Generator Assembly
OMG	Object Management Group
OOSEM	Object-Oriented Systems Engineering Method
OPM	Object-Process Methodology
PCA	Pressure Control Assembly

PCS	Portable Computer System
QFD	Quality Function Deployment
RAX	Radio Aurora Explorer
RH	Relative Humidity
RSM	Relative Supply Mass
SE	System Engineering
SMS	Space Motion Sickness
SPEs	Solar Particle Events
STM	Science Traceability Matrix
SoI	System of Interest
SoS	Systems of Systems
SysML	Systems Modelling Language
TCCS	Trace Contaminant Control Sub-assembly
THC	Temperature and Humidity Control
TTM	Technology Traceability Matrix
UML	Unified Modelling Language
UP	Urine Processor
V&V	Verification and Validation
WM	Waste Management
WP	Water Processor
WRM	Water Recovery and Management
ppO_2	Oxygen Partial Pressure

Introduction

In recent years, the design and development of space systems have been increasingly challenged by growing technical complexity, tighter development schedules, and more demanding performance and safety requirements. Modern space missions involve highly interconnected subsystems, multidisciplinary design processes, and a need for early verification of architectural decisions. Traditional document-based systems engineering approaches, while still widely used, often struggle to ensure consistency, traceability, and effective communication across disciplines, particularly during the early phases of conceptual and preliminary design.
Model-Based Systems Engineering (MBSE) has emerged as a promising methodological framework to address these challenges by shifting the focus from document-centric to model-centric system representation. In MBSE, digital models become the primary means for capturing, integrating, and managing system architecture, requirements, functions, and interfaces throughout the system life cycle, showing potential benefits in terms of improved design coherence, reduced development risk, and enhanced decision-making capabilities, especially in complex engineering domains such as space systems.
Despite the growing interest in MBSE within both academia and industry, its practical adoption remains uneven. One of the main limitations lies in the scarcity of concrete, end-to-end application examples that clearly demonstrate how MBSE tools can be effectively employed in realistic space engineering scenarios. In addition, the current MBSE tool landscape is fragmented, with different tools offering varying levels of abstraction, modelling languages, and degrees of integration with simulation environments, making the selection and application of an appropriate MBSE tool during the conceptual design phase a non-trivial task, particularly for educational and early-stage industrial contexts.

This thesis addresses these issues by investigating the potential of System Composer as an enabling tool for MBSE in space system conceptual design. The primary objective is to evaluate System Composer in comparison with other MBSE tools currently available, focusing on its capability to support the standardisation and structuring of system architectures across multiple levels of detail. To this end, the

thesis develops a conceptual design of a CubeSat mission using MBSE principles, demonstrating how System Composer can be employed to model both mission-level and system-level architectures in a consistent and scalable manner, aiming to provide a practical learning framework for the effective use of System Composer within an MBSE-oriented design process.
Beyond the CubeSat application, the thesis further explores the strengths of the tool through the digital modelling of a highly complex and safety-critical system: the Environmental Control and Life Support System (ECLSS) of the International Space Station (ISS). In this case study, the human being is explicitly treated as the central payload, and human needs are integrated into the system architecture through principles of Human-Centred Engineering, allowing the investigation of how MBSE, supported by System Composer, can facilitate the representation and management of interactions between technical subsystems and human requirements within the overall design chain.

From a methodological perspective, the work begins with a comprehensive literature review of MBSE in the scientific and space engineering domains, examining recent advancements, recognised challenges, and commonly adopted tools, followed by a practical implementation using System Composer and Simulink, initially applied to the modelling of a CubeSat mission and spacecraft, and subsequently extended to the modelling of the ISS ECLSS.

The thesis is structured as follows.
Chapter One presents the state of the art in Model-Based Systems Engineering, including a comparison between document-based and model-based approaches.
Chapter Two provides a comparative analysis of existing MBSE tools, their underlying modelling languages, and their respective strengths and limitations.
Chapter Three focuses on System Composer, analysing its functionalities, distinctive features, and limitations through an illustrative application example.
Chapter Four introduces the first case study, providing the workflow and analysis of a CubeSat mission and spacecraft.
Chapter Five introduces the principles of Human-Centred Engineering and presents the second case study focused on the ISS Environmental Control and Life Support System, demonstrating the practical application of the MBSE methodology using System Composer.

Chapter 1

Space missions and systems

Space missions, whether focused on Earth observation, scientific exploration, navigation, telecommunication, or technology demonstration, are developed within a strict and complex ecosystem. Each mission is the outcome of a long-term engineering process which purpose lays in the integration of various subsystems into a coherent and reliable space system that can satisfy the given requests and requirements.

Modern space projects operate within a highly structured engineering framework, where rigour and traceability are essential to achieve mission success. For this reason, design, verification, validation, and implementation are guided by established international standards, most commonly by the European Cooperation for Space Standardization (ECSS). These standards provide a common foundation for system engineering, quality assurance, and project management, enabling consistent practices across agencies, industry, and research institutions.

Within this structured environment, however, the increasing sophistication and rapid evolution of small satellite missions have begun to expose the limitations of traditional engineering practices.

1.1 State of Art

The limitations of traditional Document-Based Systems Engineering (DBSE) are exposed by the growing complexity of small satellite systems, where static documentation and linear processes often leads to miscommunication, inefficiencies and expensive delays that, coupled with the manual creation of artefacts like requirements, traceability matrices, and interface documents, results in inconsistent information and fragmented collaboration among stakeholders. On the other hand, Model-Based System Engineering (MBSE) promotes an integrated and model-centric approach that tries to unify requirements, design and validation within a visual framework that enable real-time update and improved traceability. While facing adoption challenges, such as training and tool costs, MBSE has the potential to modernise development and enhance collaboration for small satellite projects.

1.2 Traditional System Engineering vs MBSE

Space systems are quite different from the common ones found on ground and, by having a vast number of interactions between the wide number of subsystems, a high reliability is mandatory [1]. For this reason, obtaining the best space products require a meticulous optimisation of new methodologies that form the management and the organization of the system design process. So, managing the product life cycle is fundamental to build a correct and compliant space system, by following a series of recursive relationships between stakeholders' analysis, technical requirements, logical decomposition, and design solutions [1].

1.2.1 Document-Based System Engineering

The traditional approach to systems engineering relies on a collection of documents to manage all aspects of a project's lifecycle, from requirements to system architecture and design up to the more programmatic aspects such as cost and risk analysis. Across disciplines, documentation stays essential for communication both within and between teams, but as project complexity increases, the number of documents tends to grow rapidly, often without clearly defined dependencies. Any modification requires manual updates across multiple documents, typically written in natural language, which can compromise completeness and consistency, potentially resulting in conflicting or contradictory information [2].

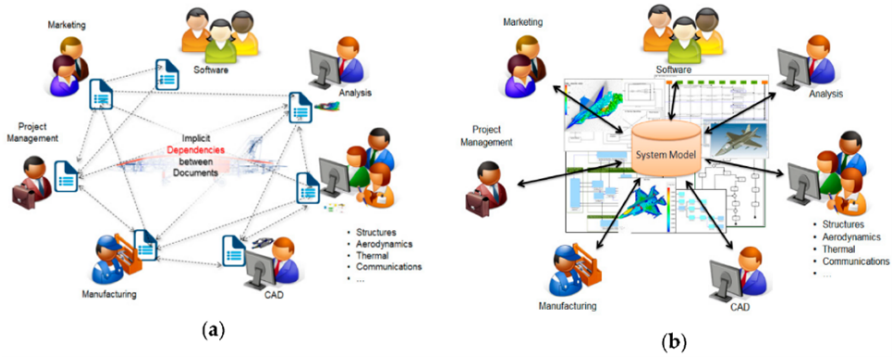


Figure 1.1: (a) DBSE communication, (b) MBSE communication [3].

In their report, Adedjouma finds several key limitations of the traditional DBSE approach. Although seen in a specific project, these issues apply to complex system development:

1. *Project Planning and Workflow.* In DBSE, documents serve as the primary artifacts for capturing requirements and design information. Project teams focus on writing, reviewing, updating, and publishing documents according to sequential workflows. Data remains static between releases, which are often scheduled according to business objectives rather than design needs. To meet deadlines, high- and low-level tasks are executed in parallel, often resulting in late document finalization and rapid obsolescence post-publication, with no availability of updated specifications between official releases.

2. *Design Semantics.* Text-based specifications, also supplemented with diagrams, are used to describe system requirements and design decisions.
   However, natural language is often inadequate for conveying complex design logic and ambiguities may arise, such as technical decisions being misinterpreted as requirements. In addition, inconsistencies between text and diagrams can lead to overspecification and unnecessary rework.

3. *Traceability.* Traceability between high- and low-level design elements is limited, particularly due to the use of non-traceable visual elements. Since traceability is tied to document releases, any changes introduced between versions are difficult to manage. This lack of continuous traceability affects the analysis and makes it challenging to estimate the time and cost of requested modifications.

4. *Communication.* Team communication relies on the publication of documents aligned with formal release schedules. Supporting materials, such as diagrams and presentations, are created on demand and, because information is distributed across multiple documents, many aspects of the system are difficult

to communicate clearly and prone to misinterpretation.

5. *Tooling*. DBSE commonly involves a suite of disconnected commercial tools (e.g., PLATO, Excel, Word, MATLAB) that lack unified integration, needing manual data transfers that are time-consuming and error prone.

6. *Standard Compliance*. Tools based on rigid, proprietary meta-models (e.g., MATLAB) are not easily adaptable to the DBSE framework, creating gaps in the development process.

These structural limitations reinforce the need to transition toward MBSE, which addresses many of the shortcomings in the document-based approach and offers a more integrated, traceable, and adaptive framework, more suited for the development of complex systems such as small satellites [4].

1.2.2 Model Based System Engineering

MBSE is the application of formalized models to support system requirements, design, analysis, verification, and validation activities from the early design phase through development and implementation.
MBSE adopts a "model-centric" paradigm, in which a system model integrates requirements, design logic, and verification data, becoming the primary artifact and source of truth for the development team, while documents are derived outputs. MBSE improves communication among stakeholders, reduces development risks, enhances design quality, and facilitates more effective knowledge management across the system lifecycle by promoting consistency and rigour in capturing and maintaining engineering information [5].
Siddique states that the shift from DBSE to MBSE is more than a methodological change, it is a strategic transition essential for organizations looking to be competitive in the evolving satellite industry.
As illustrated in a comparative analysis of DBSE and MBSE, the traditional approach is associated with several limitations [6].

Table 1.1: Comparative analysis between DBSE and MBSE [6].

DBSE limitations	**MBSE advantages**
Higher costs due to late-stage problem identification	Improved efficiency through early issue detection
Extended project timelines resulting from rigid processes	Shorter development cycles via iterative processes
Reduced stakeholder satisfaction from communication gaps	Enhanced collaboration and stakeholder alignment
Increased complexity in integrating heterogeneous components	Simplified subsystem integration
Limited flexibility in responding to changing requirements	Greater adaptability to evolving mission and technical requirements

Further analysis by Adedjouma highlights the benefits and limitations of the MBSE approach when compared with the DBSE methodology observed in their case study:

1. *Project Planning and Workflow.* A centralized system model effectively replaces traditional documents, serving as the primary environment for design and specification and, unlike the sequential document-writing process in DBSE, enables more flexible project planning, allowing engineers to work simultaneously on different abstraction levels.

2. *Design Semantics.* The use of SysML models and diagrams in place of natural language descriptions improves clarity and reduces the risk of overspecification.

3. *Traceability.* MBSE enhances traceability by autonomous tracking of relationships among system elements, while SysML includes built-in relationship types, supporting consistent justification of design choices and easing certification efforts through integrated traceability.

4. *Communication.* Stakeholder communication is improved by using a unified modelling language which supports requirement specification, architectural modelling, and behavioural representation. The ability to generate custom views and diagrams directly from the model supports diverse communication and design needs. Documentation is no longer strictly tied to release milestones but evolves continuously in response to stakeholder input. However, this shift

introduces new communication challenges, and targeted training is needed to address skills gaps and mitigate associated business risks.

5. *Tooling.* Tools have been adopted to support MBSE workflows, including plug-ins that enable synchronization between SysML and MATLAB environments.

6. *Standard Compliance.* Continuous traceability, embedded in the MBSE process through automated mechanisms during system definition and decomposition, helps bridge the gap between high-level specifications and system implementation.

These observations underline MBSE's potential to improve system development processes, while also highlighting the importance of organizational readiness, tool interoperability, and workforce training in ensuring its successful adoption [4].
The transition from DBSE to MBSE can be misleading, as it may suggest that documentation is no longer necessary within the engineering process or in stakeholder communication. However, most stakeholders involved in a project are not systems engineers and are more comfortable engaging with project information through documents, whether printed or digital. Moreover, certain model contents must be interpreted for a variety of purposes, including legal compliance or regulatory constraints. Logan argues that while an MBSE approach that excludes documentation may be effective within the project team, it shifts the burden onto stakeholders during review and implementation phases. Recognising the ongoing need for documentation, a potential solution is to use structured document forms that serve as inputs to the system model. Initial information can be elicited or updated efficiently by designing these as "fill-in" templates and the resulting data can then be linked to existing model content [7].
In addition, Calò states that MBSE methodologies can enable the separation of the project environment from the required documentation by automating report generation. Since system model diagrams can store all information relevant to each development phase, documents can be derived directly from the model through scripts that generate predefined templates, allowing engineers to focus more on modelling activities rather than on manual documentation tasks [1].
In his dissertation, Cencetti notes that integrating MBSE into environments where established methods and processes are already in place may lead to drawbacks. To avoid inferior outcomes compared to traditional procedures, the implementation of MBSE should be assessed on real-world problems. Successful integration requires early acceptance from professionals with domain-specific experience, making training a critical factor that should not be underestimated. Furthermore, the infrastructure for MBSE must be clearly defined to effectively support a model-based engineering environment.

The application of MBSE also demands significant conceptual effort to define the features and relationships among system elements throughout the lifecycle. Its adoption is influenced by the availability of proper tools to support product development and the benefits and limitations of various MBSE approaches must be carefully evaluated, as no single solution can address all challenges. However, understanding the limitations of one method compared to another can guide the selection of the most suitable direction [8].
Conversely, Henderson's review highlights the current lack of empirical evidence in the public literature to support the hypothesis that MBSE yields measurable benefits in the development of engineered systems. While numerous claims are made regarding its advantages, these are often not substantiated by objective data, nor are they consistently cited across studies. This lack of unanimity reflects a broader disagreement within the community about the primary benefits of MBSE. The diversity of perspectives and experiences further shapes how its value is perceived. However, the absence of empirical validation does not suggest that MBSE is inherently disadvantageous; rather, it underlines the need for more rigorous assessment and case-based evaluation [9].

1.3 MBSE in space system design

1.3.1 General adoption in aerospace industry

The aerospace sector is one of the most prominent industries adopting MBSE, yet the incomplete nature of its implementation has received limited scholarly attention. In their study, Pratt and Dabkowski applied the Unified Theory of Acceptance and Use of Technology (UTAUT) to identify a range of challenges and enablers associated with MBSE adoption. Areas such as transformational leadership, trust, and cost considerations were highlighted as having significant influence on the successful implementation of MBSE practices [2].
ESA has identified model-based practices as essential for achieving future objectives in engineering, procurement, and financial digitalisation. Space projects are inherently complex from a systems engineering perspective, due to their extended duration, involvement of multiple organisations, system intricacy, and financial or political constraints.
In response, ESA is shifting from a traditional document-based approach towards a more model-centric paradigm, both internally and in its interactions with stakeholders. This transition includes the development of the ESA MBSE Methodology, supported by dedicated training programmes to enable effective implementation of the approach and related tools.
Initial efforts to formalise a model-based systems engineering framework led to the development of SysML, which has since been applied in several projects and studies. Examples include the work by Raif [10], who employs SysML for the dynamic system simulation of small satellites, and the study by Calò [1], which presents an application of MBSE to small satellite systems.
The aim of the ESA MBSE Methodology is to enhance existing systems engineering processes, while keeping the flexibility required to adapt to the unique demands of individual missions. These processes, despite their project-specific customisation, share common elements defined by the ECSS standards.
The ESA MBSE Methodology is structured into five distinct layers:

1. *Mission Specification.* This layer defines the primary elements of the mission and the space system, including mission phases, objectives, and scenarios.

2. *System of Interest (SoI) Specification.* This focuses on defining the system under development.

3. *Functional Design and Physical Design.* These layers describe the system's decomposition into lower-level components or products that implement the functions defined at the functional level.

4. *Transversal Layers.* These layers introduce concepts applied across all other

layers:

- *External Systems* refer to any systems or actors that interact with the space system or SoI.
- *Exchange Items* refer to any objects, such as information, matter, or energy, exchanged between entities within the model (e.g., functions, components).

5. *Requirements Layer*. This layer captures and integrates textual requirements in the system model [11].

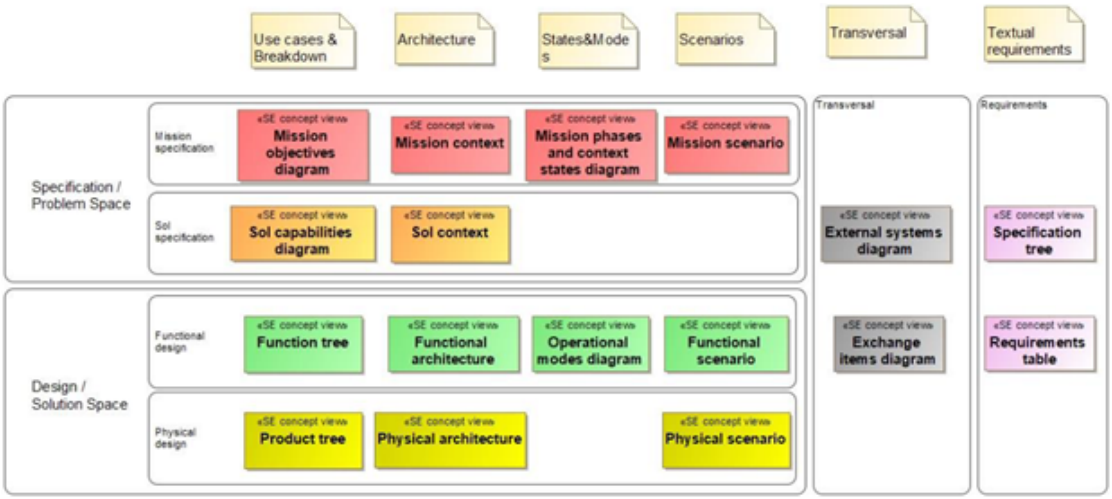


Figure 1.2: ESA MBSE Methodology structure [11]

.

It is important to emphasise that the ESA methodology does not rely on a dedicated tool; instead, SysML diagrams are used, while internally Cameo may be employed when licenses permit. Similarly, this thesis does not adopt a formally defined methodology, relying solely on the capabilities provided by MATLAB's System Composer tool.

1.3.2 Benefits for space missions

A relevant example of the benefits associated with the use of MBSE in the aerospace sector is presented by Holladay, through the application of the Model-Based Systems Engineering Infusion and Modernisation Initiative (MIAMI) at NASA. One significant case involves the MBSE Pathfinder Payload Adapter Team, which explored a model-based approach for integrating key user requirements with payload accommodation capabilities to define Payload Adapter designs tailored to each mission. By automating the development and management of requirements across the full lifecycle, from concept to as-designed and as-built configurations, and by integrating SysML models with CAD tools, manufacturing processes, and testing environments, the initiative demonstrated a reduction in both cost and design time, and also decreasing the effort required for vehicle interface definition and management activities. The adopted modelling approach included multiple interconnected models for verification and analysis, as well as a user interface linked to additive manufacturing outputs. The MBSE process model guided each step of the Payload Adapter definition process, including data initialisation, input, export, calculations, data saving, and display. Meanwhile, the SysML model defined the system's physical structure, incorporating requirements, verification constraints, and behaviours. The application of this approach resulted in several key benefits:

1. It supported communication and education about the utility of the Payload Adapter.
2. It enabled early identification of design constraints for primary, co-manifested, and secondary payloads.
3. It reduced the formal analysis effort needed after payload manifestation.
4. It helped to minimise errors arising from manual processes.
5. It saved time during prototyping and manufacturing [12].

1.4 MBSE for small satellite development

In the continuously evolving landscape of space missions and exploration, the utilisation of Model-Based Systems Engineering (MBSE) is becoming increasingly prevalent, largely due to the adaptability and flexibility offered by MBSE tools.

According to Siddique's analysis, despite the challenges associated with its adoption, the MBSE approach addresses the growing need for improved cooperation and communication among stakeholders. When combined with the typically small size of space-mission development teams, this can significantly enhance collaboration and stakeholder satisfaction. MBSE also improves requirements traceability through explicit and structured mapping, thereby supporting the formulation of clear and verifiable mission objectives. Furthermore, MBSE enables early identification of design issues, reducing overall development time, and promotes cohesive integration of heterogeneous system components by strengthening system interoperability [13].

Several practical applications of MBSE in space systems have been reported in the literature. Kaslow presents an MBSE-based integration and execution framework applied to the Radio Aurora Explorer (RAX) CubeSat mission [14].
This study proposes the development of a SysML reference model for a generic CubeSat platform, which is then implemented to model the RAX mission specifically. This reference model supports the analysis of key mission aspects, including communication downlink opportunities, power availability, and mission activities and states, and examines how these factors constrain or enable spacecraft functions. The work concludes with the development of a CubeSat enterprise model that captures cost and product lifecycle considerations for both the mission spacecraft and its broader problem domain, while also incorporating additional RAX-specific design and operational characteristics [14].

Similarly, Deiana illustrates the utility and interoperability of Data-Driven Systems Engineering (DDSE) and MBSE through the complementary use of the Valispace and Capella tools [15]. The proposed approach integrates the two methodologies to achieve a more comprehensive and consistent representation of the space system under development, while enabling efficient and dynamic data management. This integration supports the maintenance of a single source of truth, ensuring information continuity between tools and fostering effective collaboration among engineering teams and stakeholders. The thesis presents the adopted methodology, the tool integration process, and the results obtained from a small-satellite case study, demonstrating how the combined use of DDSE and MBSE optimises the design development process, mitigates risks, and enhances overall system agility [15].

In addition, in his report Anyanhun demonstrates the application of MBSE to the verification and validation of an inter-satellite communication system architecture [16]. The study emphasises the importance of adopting multiple modelling perspectives to enable effective V&V, namely the architectural framework perspective, the requirements perspective, and the system perspective. The architectural view supports the assessment of structural consistency and interface correctness, the requirements view ensures traceability and completeness of requirement implementation, and the system view enables validation of overall system behaviour under representative operational scenarios [16].

Chapter 2

Comparative analysis of MBSE Tools

The Unified Modelling Language (UML) is a standardized modelling language developed to specify, visualize, construct, and document the artifacts of a "software system" or, as clarified in UML 2.0, more generally a "system." Its customization for system engineering supports the modelling of systems that may encompass hardware, software, data, personnel, procedures, and facilities. The aim is to provide a common language for system engineering to analyse, specify, design, and verify complex systems, thereby enhancing system quality, improving interoperability between engineering tools, and bridging the semantic gap among systems, software, and other engineering disciplines.
The Object Management Group's Systems Modelling Language (OMG SysML) extends UML 2 to ease the specification, analysis, design, verification, and validation of complex systems making up hardware, software, data, personnel, procedures, and facilities. SysML supports multiple methodologies, including structured analysis and object-oriented approaches. It reuses a subset of UML 2 concepts and diagrams, augmenting them with additional diagrams and constructs tailored for systems modelling [17].

2.1 Tools for MBSE

This section is dedicated to presenting the MBSE tools most commonly used in contemporary engineering design. For each tool, a brief overview is provided, describing both the architecture and the modelling language employed.

2.1.1 Capella

Offering functionalities comparable to tools such as PowerPoint, Visio, and Excel, Capella does not rely on a standardised programming or modelling language such as SysML or UML. Instead, the tool is built around **Arcadia**, a systems engineering methodology that defines its own proprietary modelling language. This language is centred on architectural concepts, such as functions, components, interactions, functional chains, and exchanges, rather than on formal syntactic constructs.

In Capella, the language is expressed through a meta-model implemented on the Eclipse Modelling Framework (EMF), which semantically constrains the modelled elements and their relationships. This design choice reflects Arcadia's primary objective: to promote system architecture understanding and consistency among heterogeneous stakeholders, while reducing the complexity and ambiguity often associated with generic modelling languages.
In this sense, Capella does not aim to propose an alternative language to SysML, but rather a guided methodological approach, in which the model itself becomes the primary means of expression.

As an open-source platform, Capella benefits from over a decade of practical feedback derived from applying the Arcadia method and its predecessor tool, Melody Advance, in Thales projects. The tool follows a structured release cycle, with a major version introducing new functionalities each year and several minor releases throughout the year to address bugs and deliver incremental improvements [18].

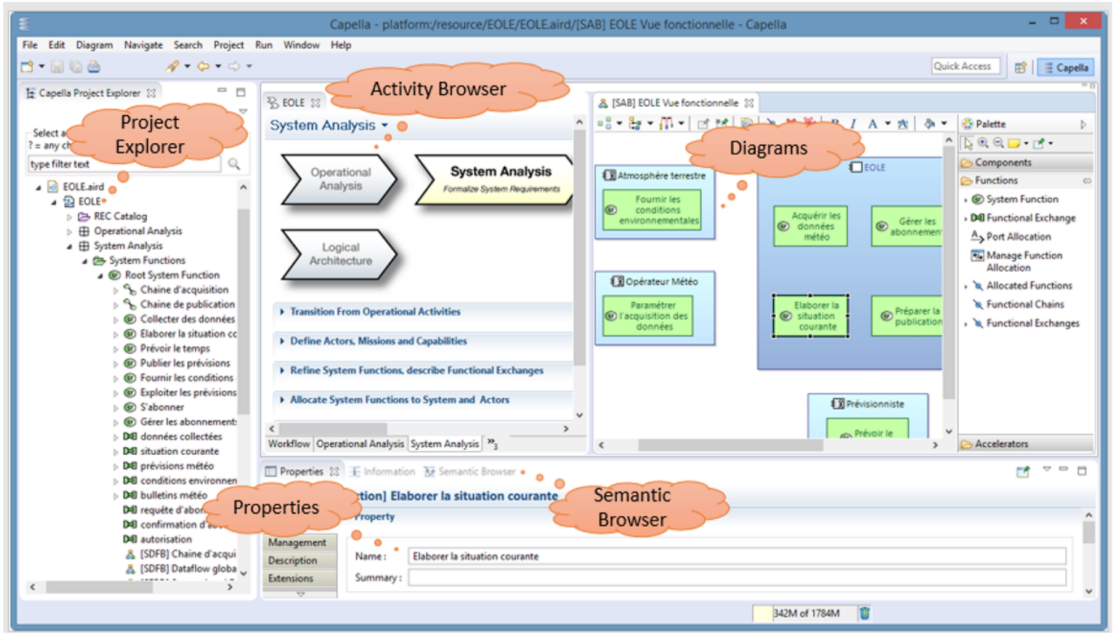


Figure 2.1: Overview of the Capella interface [18].

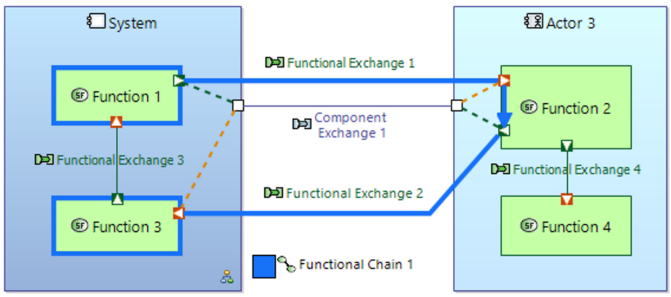


Figure 2.2: Simple example of an Architecture diagram at the System level (SAB), with a Functional Chain [18].

2.1.2 Cameo System Modeler

Cameo System Modeler (CSM) is a flagship platform for modelling complex systems within a collaborative environment. It is intuitive, robust, and among the most compatible tools with the standard SysML, which is adopted a its formal modelling language, in accordance with the MBSE approach described by Casse [19].

Like other SysML tools, it offers import and export capabilities, ensuring long-term usability. CSM is available on multiple platforms, including a portable version that requires no installation. Designed to support system engineers in efficiently creating SysML models, it also aligns with broader MBSE practices. Its functionality can be extended through plug-ins that enable simulation, remote model sharing, product line management, and variant handling. Additional plug-ins facilitate integration into existing toolchains, ensuring interoperability with third-party applications [19].

As emphasised by Casse, Cameo does not enforce a specific MBSE methodology: the tool implements the SysML language and ensures syntactic and semantic consistency, but the effectiveness of model-based systems engineering ultimately depends on the set of rules, conventions, and processes defined by the organisation or the project. From this perspective, Cameo acts as an MBSE implementation platform, while SysML represents the common language that enables communication, integration, and traceability among stakeholders [19].

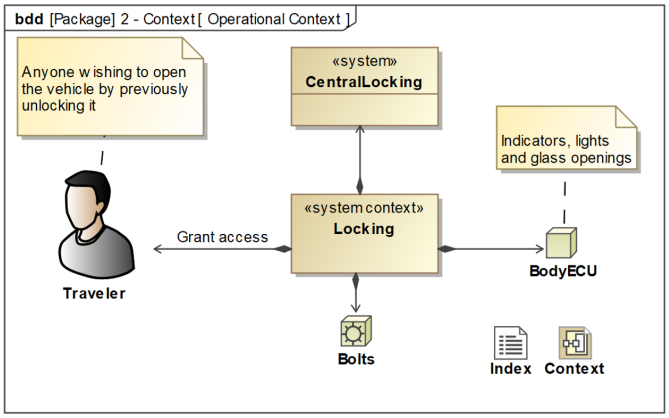


Figure 2.3: Example of a Bdd operational context diagram [19].

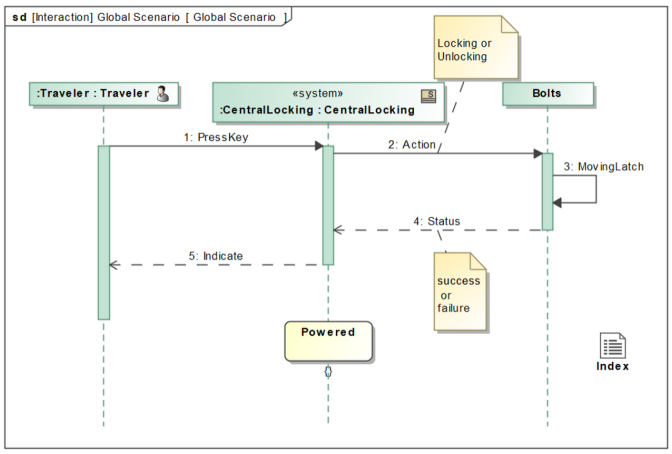


Figure 2.4: Example of a description of overall scenario [19].

2.1.3 System Composer

System Composer allows engineers to capture the early concepts and system architecture along with the design of the systems in a single tool. The aspects of the abstract system architecture work that can be directly applied and connected to detailed design work are the first focus of System Composer. There are two important aspects of System Composer that enable it to effectively work for both system engineers early in the development process and design engineers who work in Simulink.
First, it is easy and intuitive for system engineers to begin their work of abstract modelling of architectures and functional decompositions.
Second, it is tightly coupled to the Simulink design environment [20].

Watkins highlights that System Composer does not rely on an independent standard modelling language such as SysML. Instead, it employs an internal architectural representation composed of elements such as components, ports, interfaces, and connections. The language is not defined by a standardised formal grammar but by a proprietary meta-model integrated within the MATLAB/Simulink environment, which allows high-level architectures to be described and directly linked to simulated behaviours in Simulink or Stateflow. Architectural elements can be extended and customised through stereotypes and profiles, but the semantics remain tied to the internal structure of the tool and its integration with the engineering design environment [20].

System Composer models begin as an early way to capture concepts and mature into systems integration models that describe the integration between subsystem models. For configurable software systems, the System Composer model can be interfaced with specialized design tools to directly define the configurations, enjoying a Model-Based Design approach beyond what is achievable in Simulink alone [20].

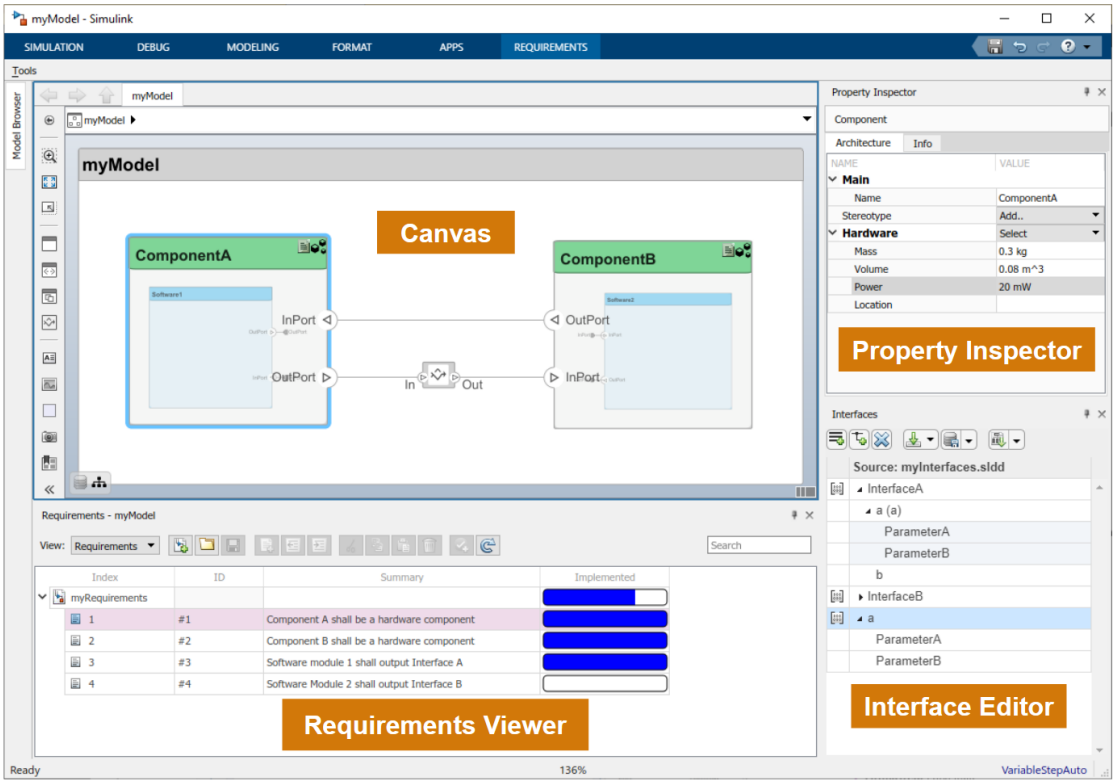


Figure 2.5: System Composer user interface [20].

2.1.4 Other tools

SCADE

Safety Critical Application Development Environment (SCADE) is both a notation and a toolset designed for the development of safety-critical systems in domains such as railways, aeronautics, and industrial applications, where compliance with certification standards is essential. SCADE notation incorporates nested state machines and block diagrams, enabling explicit initialization of data flows, clock-based time management, data dependency concurrency, and deterministic execution. SCADE Suite has been further extended with SCADE Display, a flexible graphics design and code generation toolset for safety-critical embedded display systems. Both SCADE Suite and SCADE Display provide advanced features such as fine-grained traceability, automated consistency checks and documentation generation [21].

Specifically, SCADE Architect is built upon a SysML-based modelling environment for describing system architecture and requirements, while abstracting the complexity of standard syntax to provide a more intuitive modelling experience for users. For the design of safe and critical embedded software, SCADE Suite employs the SCADE language, a formal domain-specific language for real-time systems based on principles of synchronization, data flows, and state machines. This language is derived from synchronous languages such as Lustre and possesses a precise semantics aimed at the generation of certifiable code [21].

In addition, SCADE System offers a development environment grounded in OMG SysML, supported by open technologies (e.g. Eclipse), and fully integrated within the SCADE Suite framework. It synchronizes system design with SCADE Suite and SCADE Display while sharing their traceability and documentation tools, thereby ensuring consistency across the entire development process [21].

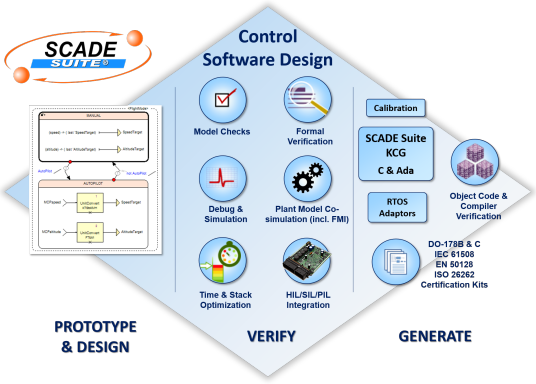


Figure 2.6: SCADE Suite Control Software Design [22].

TASTE

The ASSERT Set of Tools for Engineering (TASTE) is an open-source toolchain designed for the development of embedded real-time systems, supporting both modelling and deployment of distributed architectures.
Its primary goal is to automate repetitive and error-prone tasks that hinder the integration of complex systems [23].

TASTE relies on two lightweight modelling languages, AADL and ASN.1, while allowing system subcomponents to be implemented in C, Ada, SDL (via RTDS or ObjectGEODE), SCADE, Simulink, VHDL, or combinations thereof. Without introducing significant overhead, TASTE generates binaries that can be executed on a variety of supported targets, including native Linux, real-time Linux (Xenomai), Leon2/RTEMS, and Leon2/ORK [23].

TASTE is designed for extensibility, offering, for example, dedicated modes that enable interaction with generated binaries through Python scripts or interactive user interfaces. Unlike many modelling environments that impose a single paradigm, TASTE is not monolithic; instead, it consists of independent components that can be used cohesively within its ecosystem or separately in alternative development environments [23].

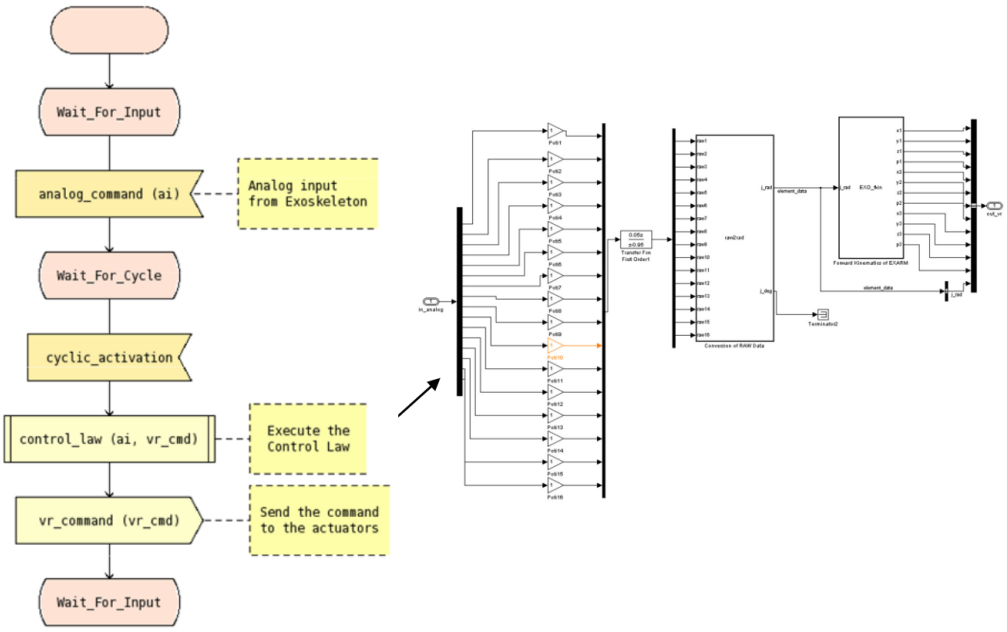


Figure 2.7: TASTE SDL-Simulink combination to model system behaviour [23].

The advantages and limitations of each tool are summarized in the following table:

Table 2.1: Comparison of MBSE Tools: Pros and Cons.

Tool	**Pros**	**Cons**
Capella	Open-source, user-friendly, strong architectural support	Lacks full SysML support, limited integration with simulation tools
Cameo System Modeler	Full SysML compliance, powerful modelling features, strong industry use	Steep learning curve, expensive, requires specialized training
System Composer	Native integration with MATLAB/Simulink, intuitive interface, ideal for MathWorks users	Not a full SysML tool, less mature for architectural modelling
SCADE	Certification-ready, integrates with MBSE environments, industry adoption	Increases workflow complexity, expensive, steep learning curve
TASTE	Tailored for space systems, open-source and free, multi-language integration, automatic code generation	Niche adoption, learning curve, not a full MBSE system

2.2 Why move toward System Composer?

"The goals of System Composer are to make system modelling easy, flexible, and scalable; and to ease the transition to the design environment. The features and constructs of System Composer reflect the prioritization of these goals" [20].
System Composer is a strong choice for the design of SmallSat space systems within an MBSE philosophy because it directly bridges system architecture and detailed engineering analysis in a single environment. Its native integration with Simulink and Simscape allows engineers to move seamlessly from requirements and architecture modelling to dynamic simulation and trade-off studies, which is essential for capturing the multidisciplinary interactions typical of satellites. Compared to heavier, compliance-driven tools, System Composer offers a lighter, more agile approach that supports rapid iteration, early validation, and cost-effective development, qualities that align well with the fast timelines and constrained resources of SmallSat missions.

Integration benefits

System Composer provides a unified environment for system architecture modelling and simulation by working natively with Simulink. This integration ensures that structural and behavioural aspects of the SmallSat design remain consistent, allowing engineers to trace architectures directly into simulation models without exporting or duplicating work across tools.

Workflow alignment

The tool supports seamless data exchange between system models, requirements management tools, and performance analysis platforms, making it easier to keep a digital thread. This alignment ensures that updates in architecture or requirements automatically propagate into analysis workflows, reducing errors and improving traceability.

Ease of adoption

For teams already using MATLAB and Simulink, System Composer offers a familiar interface and ecosystem, lowering the learning curve. Engineers can leverage their existing modelling expertise while extending capabilities into MBSE, minimizing the need for disruptive changes in processes or training.

Agility for SmallSat programs

SmallSat missions demand fast prototyping, rapid iteration, and early validation under tight budget and schedule constraints. System Composer supports this agility by enabling quick architecture trade studies, integration with hardware-in-the-loop testing, and validation of system behaviours early in the design process.

Limitations and mitigations

While System Composer does not fully implement standard SysML notation, this limitation can be mitigated by using it alongside complementary tools such as Capella or Cameo for compliance-driven documentation. In practice, its lightweight approach is an advantage for SmallSat programs, and gaps can be filled with external requirement management or standards-compliant modelling where needed.

2.3 Conclusions and Future Perspective

MBSE has come into view as an approach for the development of space systems and small satellites. By shifting from document-centric processes to model-centric practices, it enables improved consistency, traceability, and communication across engineering teams. For small-sat projects, which are typically constrained by short development timelines, limited budgets, and small teams, MBSE provides significant advantages: it promotes early detection of design flaws, ensures alignment between requirements and implementation, facilitates integration of heterogeneous subsystems, and grant more adaptability to evolving mission and technical requirements. This systematic approach not only reduces technical and managerial risks but also accelerates design iterations, allowing small-sat missions to keep agility without compromising reliability. Despite these benefits, the effectiveness of MBSE depends heavily on the choice of proper tools. Different platforms vary in their modelling paradigms, interoperability, and support for standards such as SysML. Therefore, tool choice must be weighted by the specific mission needs, including system complexity, the required level of certification, integration with simulation environments, and team expertise. A poorly aligned tool may introduce unnecessary overhead or limit flexibility, undermining the efficiency gains MBSE promises. In this context, MATLAB's System Composer is a strategic option for agile small-satellite development and its seamless integration with Simulink and the wider MATLAB ecosystem allows engineers to progress smoothly from architecture definition to dynamic simulation and implementation. This integration ends the need to manage multiple disconnected tools, reducing workflow friction and supporting rapid prototyping. Furthermore, System Composer offers a relatively low barrier to entry compared to more complex SysML-based platforms, while still

ensuring compatibility with emerging standards. For small-sat teams that rely on agility, fast iteration, and rigorous system analysis, System Composer provides a balanced combination of accessibility, robustness, and simulation-driven design, making it a compelling choice within the MBSE landscape.

Chapter 3

Working with System Composer

The following chapter is dedicated to explaining the operation of System Composer and its application to the objectives of this thesis within a MBSE framework. Particular emphasis is placed on evaluating the strengths and limitations of the tool in each stage of designing a hypothetical system. This includes an analysis of its capabilities in system decomposition, interface definition, and behaviour modelling, as well as its integration with other MATLAB and Simulink functionalities.

The chapter also highlights practical considerations encountered during the modelling process, such as the ease of use, flexibility, and constraints imposed by the tool, providing a comprehensive perspective on its suitability for supporting MBSE activities in complex engineering projects.

3.1 Modelling Stereotypes interactions

One of the most prominent features is the ability to assign *stereotypes* to different blocks. As illustrated in Figure 3.1, defining stereotypes allows the specification of internal properties, which are ultimately added as metadata to the relevant blocks or their ports. This enhances the semantic richness of the model and supports traceability.

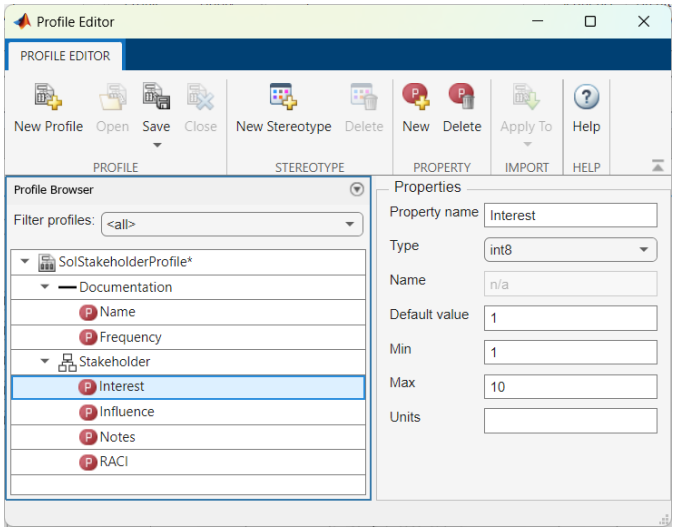


Figure 3.1: System Composer Profile Editor.

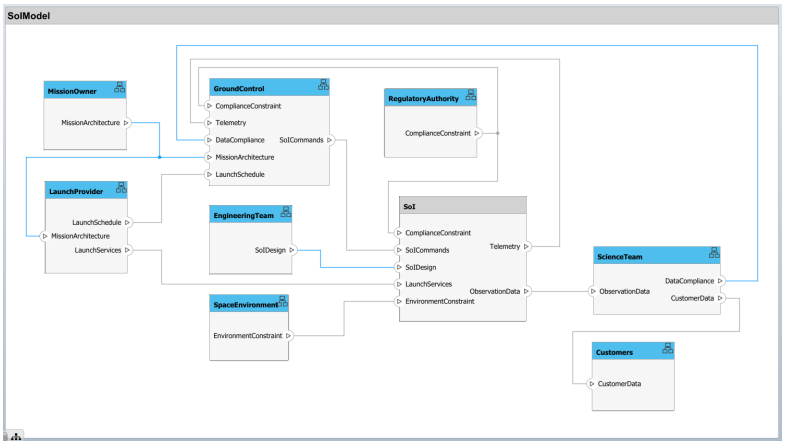


Figure 3.2: Example of stakeholders-SoI interaction architecture.

Additionally, stereotypes can be visually distinguished using symbols and colours. This proves especially beneficial in complex architectures, improving readability and aiding comprehension.

3.1.1 Limitation of Profiles and Stereotypes

Stereotypes within System Composer provide a powerful means of enriching architectural components with structured metadata. They allow the modeller to attach custom properties such as mass, frequency, link margin, power consumption, or thermal coefficients directly to elements of the architecture. This approach not only supports requirement allocation and architectural analysis but also enforces consistency and standardisation across a project, enabling engineers to treat the architecture as a single source of truth for key system parameters. The ability to define property types, default values, and units further strengthens the role of stereotypes as a semantic backbone for the model.

However, despite their advantages, stereotypes are inherently static descriptors and are not directly exploitable as signals within the behavioural simulation. A common difficulty arises when a property defined on Component A must be used as an input for Component B. Within the current toolchain, there is no direct mechanism to propagate the stereotype value across a connection, particularly in deeply nested architectures where data paths must traverse multiple hierarchical levels. The only viable approach is to manually construct data buses, define appropriate interfaces, and explicitly map the property value into a Simulink signal within the behaviour of Component A before routing it to Component B. This workflow introduces significant redundancy, since the same information must first be entered as a stereotype property and then fetched via MATLAB coding and/or functions that create a workspace of available variables.

For complex systems, this increases modelling effort and lengthens verification cycles, as every modification to a stereotype must be mirrored manually in the signal definitions. Moreover, incorrect interface specifications or mismatched bus definitions may result in compilation errors or simulation failures, which can be challenging to diagnose. Consequently, while stereotypes are invaluable for documentation and static analysis, their limited integration with behavioural modelling can impede efficiency and undermine one of the key goals of model-based systems engineering: maintaining a single, coherent representation of system knowledge.

3.1.2 Limitations of *Interface Editor*

An initial approach involved using the *Interface Editor* to define all information exchanged via connections. However, this method proved partially ineffective: while numeric data types (e.g., `int8/16/32/64`, `single`, `double`, `boolean`) can be specified, string-based data is not supported. This limitation restricts the representation of non-numeric information, such as documentation identifiers or descriptive labels.

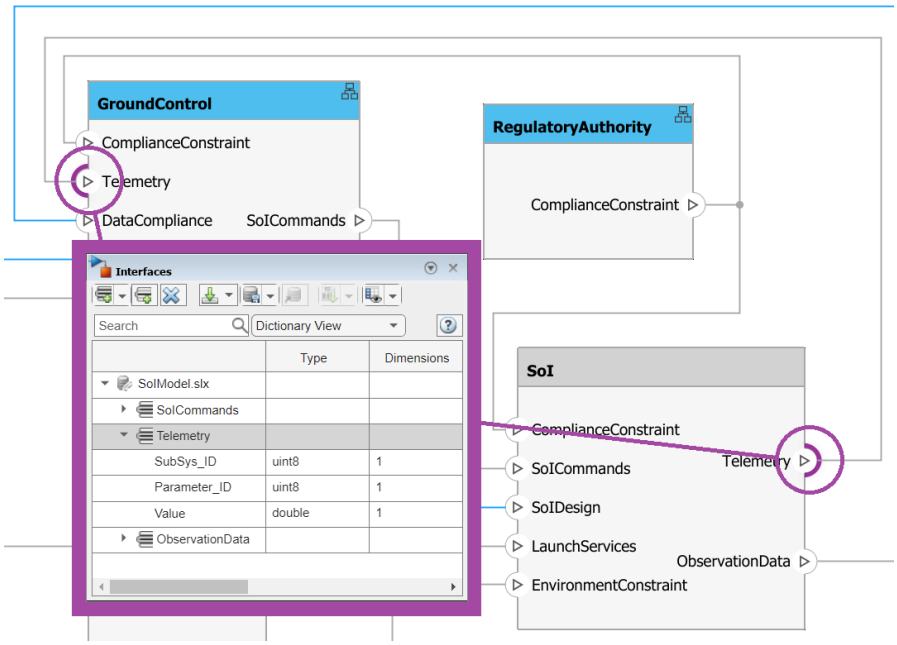


Figure 3.3: Example of *Interface Editor* output.

To overcome this, properties can be used to incorporate character strings, enabling the identification of specific documentation or data exchanged between stakeholders. A practical enhancement would be to create an external table listing all relevant documents, each with a unique code.
This allows the transmitted document to be referenced via its code, thereby integrating more effectively with the *Interface Editor*.

Connectors (i.e., the lines representing links between blocks) can also be assigned stereotypes with associated properties. However, these cannot currently be filtered within the *Architecture View*, which limits the ability to categorise and visually manage them in complex models.

3.1.3 *Architecture View* utilization

One of the principal features of System Composer is its ability to generate a well-defined representation of the entire system architecture. Through the use of the *Architecture View*, users can selectively filter components based on criteria such as name, port configuration, or internal variables. As illustrated in Figure 3.4, the components are interconnected and display their associated metadata adjacent to the output ports. A notable advantage of this functionality is the ability to edit metadata directly within the view, thereby eliminating the need to navigate repeatedly between the view and the main interface.

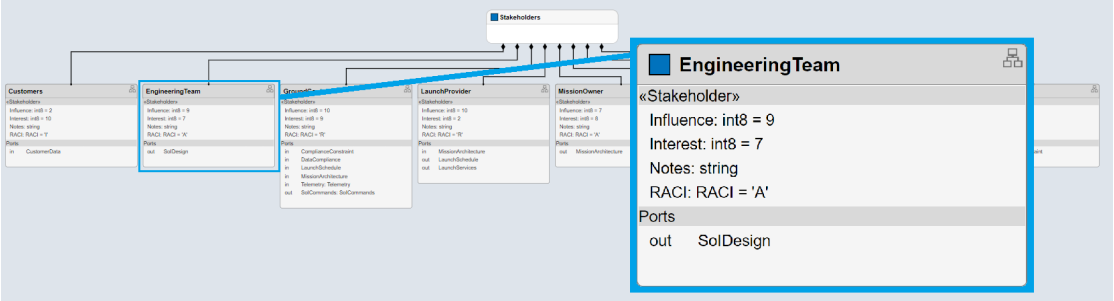


Figure 3.4: Example of *Architecture View* output.

3.2 Modelling Concept of Operations

A well-developed Concept of Operations (ConOps) is critical for the success of any space mission, as it serves as the foundational document aligning stakeholders, defining mission objectives, and describing system-level functionality. By clearly articulating how the system will operate across all mission phases, a ConOps reduces the risk of costly design changes, improves communication between teams, and ensures that requirements traceability is maintained throughout development. This structured approach enables more robust decision-making and helps guarantee that the final system meets user needs and mission goals effectively [24].

In System Composer, the *Sequence Diagram* serves as an essential instrument for illustrating and verifying timed operational sequences within a system architecture. Significantly, this diagram remains continuously synchronised with the primary architecture model, ensuring that any alterations to one are immediately reflected in the other [25]. Each element of the system can be depicted as a *lifeline*, and messaging arrows illustrate the exchange of data between components, whether signal- or message-based [26].
To articulate more sophisticated interaction patterns, such as loops, conditions, or parallel flows, Sequence Diagrams offer fragments like *Loop*, *Alt*, and *Par* [27]. Moreover, System Composer enables a co-creation workflow: lifelines, connectors, ports, and even architectural elements can be authored directly within the Sequence Diagram, with all edits maintained in synchrony with the overarching architectural model , as seen in Figure 3.5 [28].

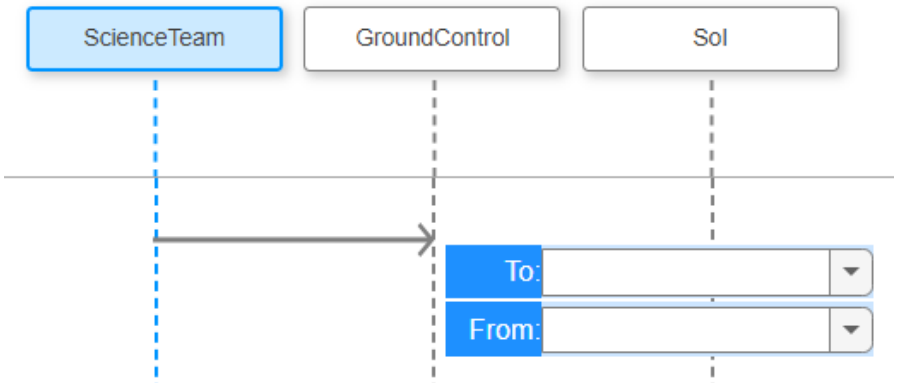


Figure 3.5: View of a connection between to lifelines.

Building on the main example, and with minor adjustments to the terminology, it is possible to construct a timed sequence that can be used to represent loops or switch cases within the Concept of Operations.

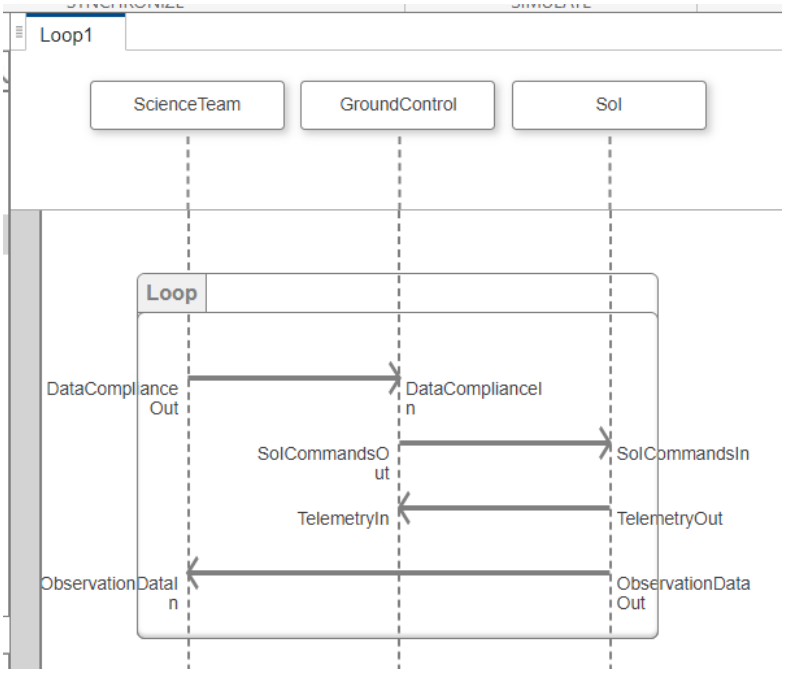


Figure 3.6: Example of a *Sequence Diagram*.

3.2.1 Practical limits of *Sequence Diagram* for ConOps

Nevertheless, practical experience with the sequence diagram highlights several significant limitations that become increasingly evident as system complexity grows. Although it is well suited to portraying a small number of components interacting in a relatively linear fashion, the visual clarity of the diagram rapidly degrades when attempting even little more complex system behaviour. This is exacerbated by the presence of control constructs such as *Loop*, *Alt*, and *Par* and other combined fragments, which, while theoretically useful for modelling conditional behaviour or iteration, introduce substantial visual clutter. Once inserted, these control elements cannot be easily removed or simplified, thereby locking the modeller into a more rigid and cumbersome representation. In a large system, this often leads to diagrams that are unwieldy, difficult to interpret, and prone to miscommunication between team members, thereby undermining one of the primary purposes of the notation: to improve understanding.
Another practical limitation lies in the manner in which the messages themselves must be annotated. In System Composer, the descriptive labels for each message arrow must be added manually as independent text boxes. This approach not only introduces additional overhead for the modeller but also opens the door to inconsistencies and potential traceability issues. Because these textual annotations are not formally linked to the underlying architectural elements or interfaces, there

is no guarantee that they remain up to date when the model evolves.
This manual process is thus inherently fragile and risks breaking the rigorous traceability that MBSE seeks to enforce between requirements, architecture, and verification activities.
These challenges reveal an underlying assumption in the design of the sequence diagram formalism: it was originally intended for small-scale systems or software-intensive designs where message exchanges can be closely coupled to code implementations. Indeed, its integration with executable notations such as Stateflow reinforces this view, since more complex logic or verification routines can only be expressed through embedded code in Stateflow charts rather than directly within the sequence diagram itself. As a result, when working within System Composer for MBSE, the sequence diagram serves more as a descriptive or illustrative tool than as a mechanism for executable specification or comprehensive verification.
Given this constraint, the primary verification capability available to the modeller is a consistency check, which ensures that all lifelines present in the diagram participate in at least one message exchange. While this is a valuable structural verification step, it falls short of providing behavioural verification or requirement compliance assessment. The result is that sequence diagrams, while conceptually powerful for communication and stakeholder engagement, are of limited use for large-scale, model-driven verification activities and should be employed with care, ideally reserved for scenarios where a compact, comprehensible interaction narrative is sufficient.

3.3 Modelling Requirements

In the context of space missions, the definition and management of requirements are fundamental to ensuring mission success. Requirements serve as the formal expression of stakeholder needs and mission objectives, guiding the design, development, and verification of complex aerospace systems. They provide a structured framework that enables traceability, facilitates communication among multidisciplinary teams, and mitigates risks associated with ambiguity or scope creep. As Douglas note, requirements should be quantitative, verifiable, and free from prescriptive solutions, thereby enabling flexibility in design while maintaining alignment with mission goals [29].

3.3.1 Requirement Toolbox

As shown in Figure 3.7, using the *Requirements Toolbox*, requirements can be created directly inside System Composer. These requirements can include custom attributes, hierarchical organisation, descriptions, and acceptance criteria. Through the *Requirements Editor* it's possible to define, edit, and classify requirements and maintain them within the MathWorks ecosystem [30].

As shown in Figure 3.7, one of the most robust and integrated approaches is to use the *Requirements Toolbox* to create requirements directly within System Composer. This workflow allows to define requirements in a structured way, including the use of attributes, descriptive text, acceptance criteria, and hierarchical organisation. Once created, these requirements are immediately available for linking to components, ports, and interfaces in the architecture model, as seen in Figure 3.12.
The direct integration facilitates traceability and enables the establishment of a digital thread from system-level requirements down to detailed architecture elements. Another advantage of this approach is the possibility of performing formal verification and validation activities within the same environment, since requirements are stored natively and can be traced to tests and simulations.

However, this method is not without its drawbacks. Care must be taken to avoid ambiguous or redundant requirements, since poor organisation can undermine the benefits of traceability. Finally, when collaboration with external tools or partners is required, working solely within MATLAB might make integration more difficult unless a consistent import/export strategy is defined.

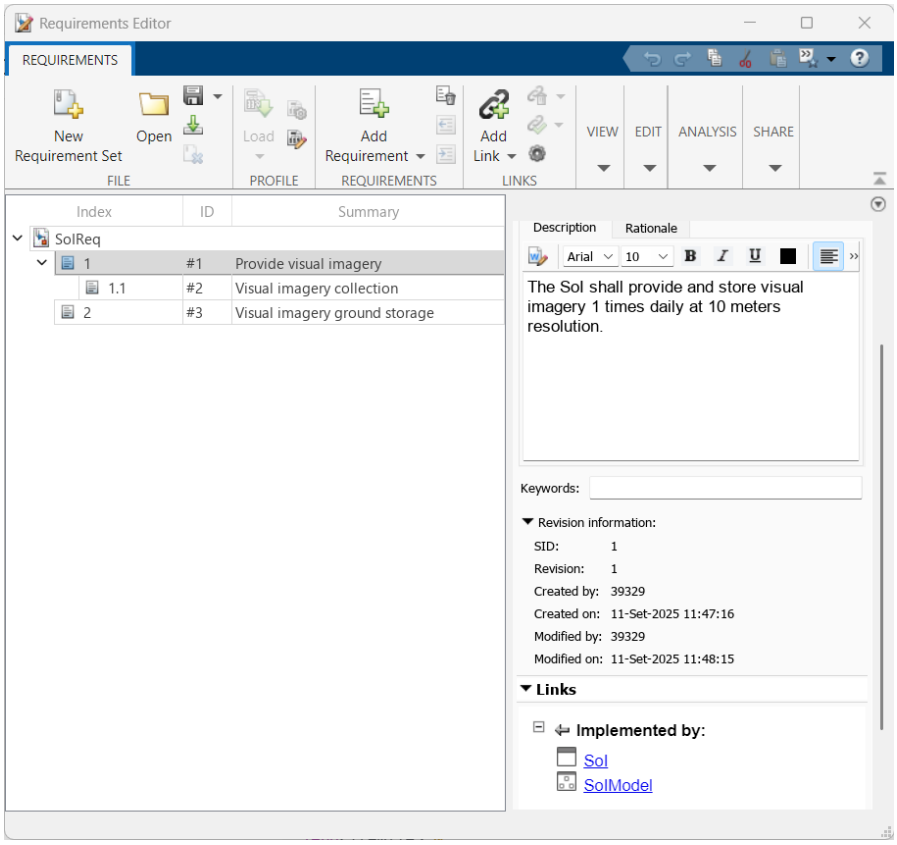


Figure 3.7: Example of *Requirement Editor*.

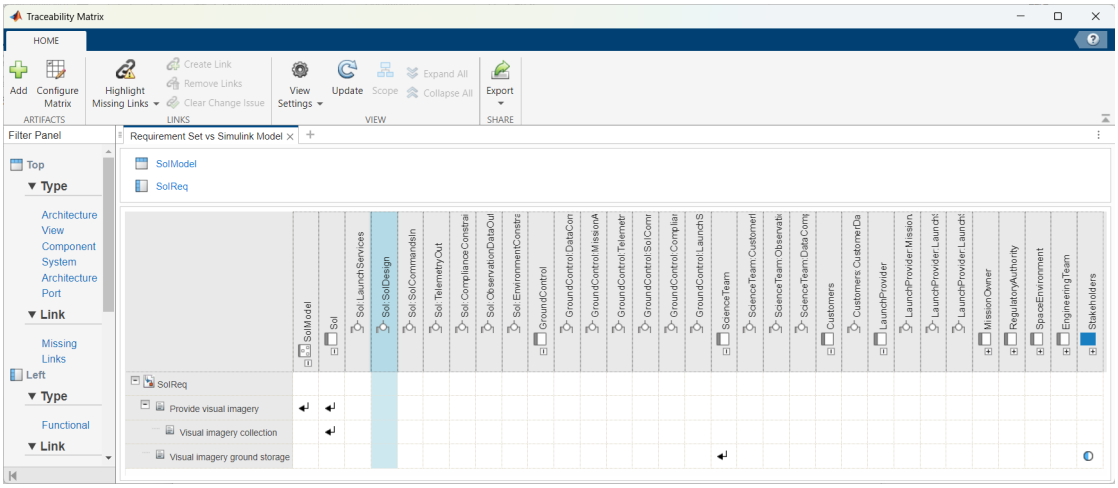
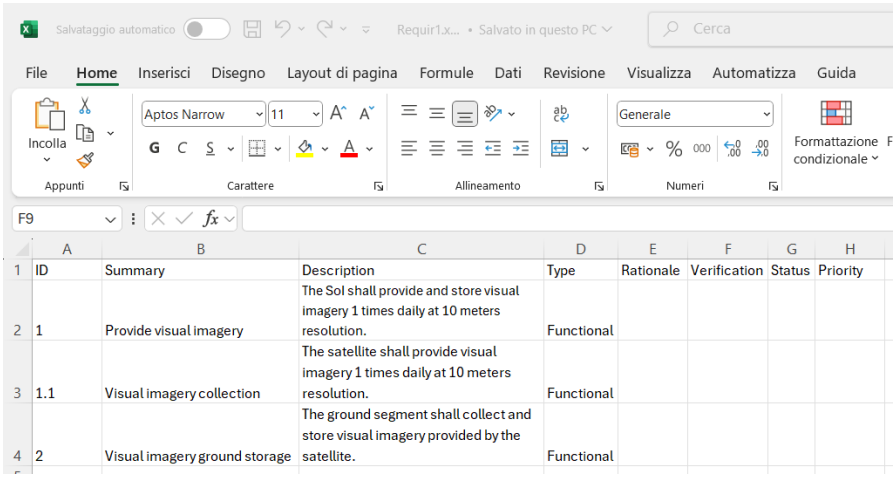


Figure 3.8: Example of an early Traceability Matrix.

3.3.2 Importing Requirements from external sources

Another frequently used approach consists of importing requirements from external documents or requirement management tools. System Composer, via the Requirements Toolbox, supports the import of requirements from Microsoft Excel, Microsoft Word, or through the industry-standard ReqIF format. This allows teams to reuse customer specifications or previously defined requirements without rewriting them from scratch. The import process can preserve hierarchy, attributes, and unique identifiers, ensuring that the requirements remain traceable. Once imported, they can be linked to architecture elements in the same way as natively created requirements. This method is particularly advantageous when collaborating with external stakeholders or when requirements are delivered in an agreed-upon document format. Nonetheless, if the source documents are poorly structured or inconsistently formatted, the import process may lead to information loss or require significant manual clean-up.
Additionally, because external requirements may change over time, maintaining synchronisation between the external source and the System Composer model can require significant manual effort, unless an automated synchronisation workflow is established [30].



	A	B	C	D	E	F	G	H
1	ID	Summary	Description	Type	Rationale	Verification	Status	Priority
2	1	Provide visual imagery	The SoI shall provide and store visual imagery 1 times daily at 10 meters resolution.	Functional				
3	1.1	Visual imagery collection	The satellite shall provide visual imagery 1 times daily at 10 meters resolution.	Functional				
4	2	Visual imagery ground storage	The ground segment shall collect and store visual imagery provided by the satellite.	Functional				

Figure 3.9: Excel document definition.

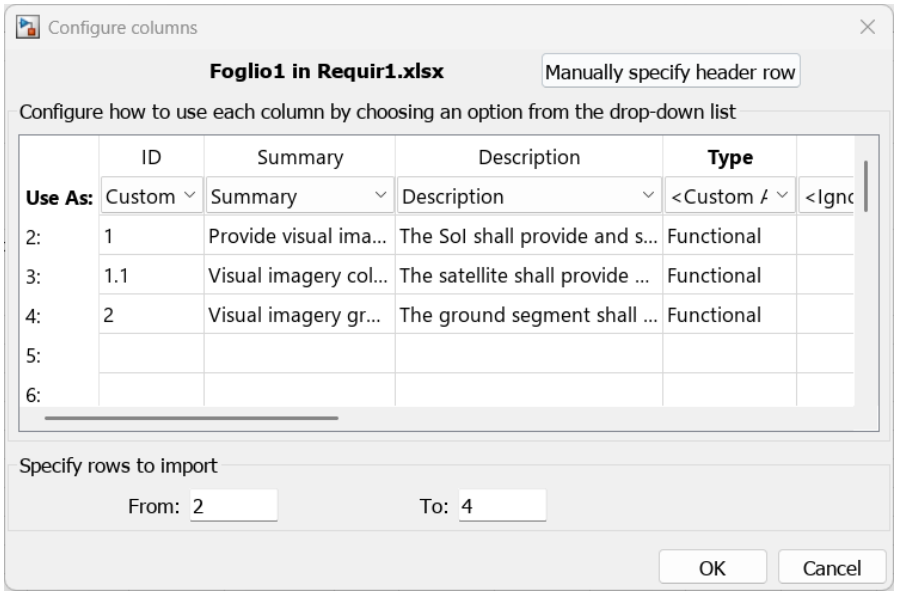


Figure 3.10: Importing requirements from Excel.

Figure 3.11: Imported requirements.

3.3.3 Linking Requirements to architecture elements

Regardless of whether requirements are created natively or imported, a critical step is to link them to the elements of the architecture. This process can be carried out using the Requirements Perspective, context menus, or the Requirements Editor, and allows each requirement to be formally associated with one or more components, ports, or interfaces. The main benefit of this linking activity is that it provides clear traceability between requirements and the design solution, thus enabling engineers to perform coverage analysis and to understand which parts of the architecture are affected when a requirement changes.
In addition, System Composer can display visual badges directly on the architecture diagram, showing at a glance which elements are associated with requirements, which is helpful during design reviews and traceability audits. Despite these advantages, there are some caveats. Linking is only meaningful when requirements are precise and verifiable; vague or overly generic requirements may lead to ambiguous

or arbitrary associations.
Moreover, in very large models, the number of links can become difficult to manage and maintain, and engineers must invest effort in ensuring that link coverage remains accurate and relevant.

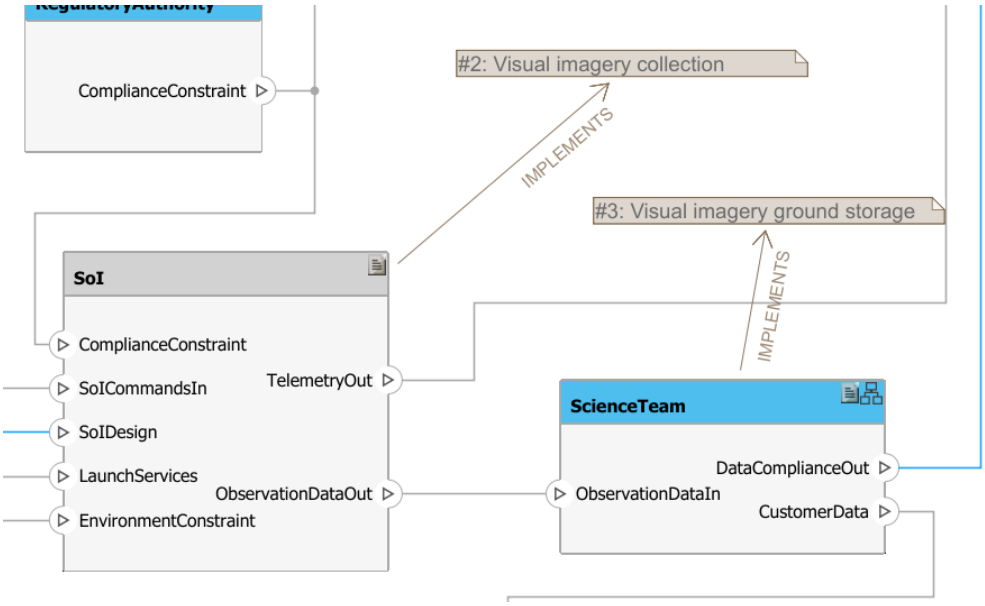


Figure 3.12: Example of linking Components with Requirements.

System Composer offers a set of visualisation tools, such as views, annotations, and badges, that make it easier to understand the requirement coverage within the architecture model. Badges can be displayed directly on components to indicate that they are linked to one or more requirements, and annotations can provide additional descriptive text or references. This visual approach is particularly valuable during design reviews, as it provides stakeholders with an immediate, intuitive understanding of which architectural elements satisfy which requirements. It also improves communication between team members by making requirement coverage visible within the graphical model.
Nevertheless, these visual elements can also become a source of clutter in complex architectures, particularly when many requirements are linked to many elements, resulting in a diagram that is difficult to read. Maintaining the consistency of these visual cues also requires discipline; if annotations are not kept up to date, they can create confusion rather than clarity [30].

3.3.4 Alternative and less formal approaches

In some cases, one may opt for less formal methods of capturing requirements, particularly during early stages of concept development or rapid prototyping.
For instance, requirements can be written as free-text comments, stored as custom properties in components, or simply described in model documentation without using formal linking mechanisms. This approach has the advantage of being very quick and does not require any additional toolboxes, making it accessible to all users. However, it lacks the rigour of formal traceability, making it easy to lose track of which requirements are satisfied and to what extent.
Another alternative is to manage requirements entirely outside MATLAB, using third-party tools such as DOORS or Jama, and simply referencing them manually in the model (e.g., by including their identifiers in component descriptions). While this allows organisations to leverage existing requirement management infrastructure, it introduces the risk of divergence between the external repository and the model, as synchronisation must be done manually.
In some cases, it's possible to embed requirements directly as architectural elements, for example by creating dedicated components named after requirements. Although this makes requirements visible in the diagram, it blurs the distinction between specification and design and can lead to duplication and inconsistency. For these reasons, such informal approaches should be considered temporary solutions, to be phased out once a more formal requirements management process is adopted.

3.3.5 Requirements verification

Verification is achieved by associating each requirement with simulation-based test cases, formal properties, or analysis results created in *Simulink Test* or other verification tools. This linkage enables automated consistency checks and facilitates impact analysis by allowing engineers to determine whether modifications in the architecture compromise requirement fulfilment. The resulting verification status, pass, fail, or untested, is displayed directly within the model, providing a model-based approach to validating system compliance prior to implementation [30].

For the sake of clarity, an illustrative example of requirement verification is presented through the analysis of a simplified link budget computation, implemented in the MATLAB function *linkBudget*. In this study, the function calculates the link margin in decibels as the difference between the achieved energy-per-bit to noise spectral density ratio, E_b/N_0, and the specified minimum required value. The inputs to the function, which are deliberately constrained via *Constant* blocks, including:

- power of the transmitter in Watts (Pt)
- transmitter gain in decibels (Gr_dB)

- data rate of the transmission (*dataRate*)
- receiver antenna diameter (*D*)
- frequency in GHz (*f_GHz*)
- propagation loss in decibels (*L_path_dB*)
- atmospheric loss in decibels (*L_atm_dB*)
- pointing loss in decibels (*L_point_dB*)
- line loss in decibels (*L_line_dB*)
- system noise temperature in decibels (*Tsys_dB*)

```
1  function margin_dB  = linkBudget (Pt, Gt_dB, dataRate, D, f_GHz,
       L_path_dB , L_atm_dB ,L_point_dB , L_line_dB , Tsys_dB)
2
3  Eb_N0_req = 14; %from QPSK modulation
4
5  Pt_dB = 10*log10(Pt);
6  Gr_dB = 20.4 + 20*log10(D) + 20*log10(f_GHz);
7
8  EIRP = Pt_dB + Gt_dB - L_line_dB;
9  L_tot = L_path_dB + L_atm_dB + L_point_dB;
10 Eb_N0 = EIRP - L_tot + Gr_dB + 228.6 - Tsys_dB - 10*log10(dataRate
       ) ;
11
12  margin_dB = Eb_N0 - Eb_N0_req ;
13
14  end
```

Within the function, the total link loss is computed as the sum of the individual losses, after which the received E_b/N_0 is calculated by combining the transmitted power, total losses, receiver gain, a constant term accounting for the Boltzmann constant and unit conversions, and a logarithmic correction based on the system temperature and range. The resulting link margin, expressed in decibels, is then obtained as the difference between the calculated E_b/N_0 and the required threshold value, thereby providing a quantitative measure of link performance against the specified requirement.

Once the *linkBudget* function is integrated into the subsystem architecture, it is placed within a Simulink Test harness, which provides a controlled simulation environment. Inputs are defined using *Constant* blocks to ensure deterministic and repeatable results, while enabling systematic evaluation. The output *margin_dB*

is connected to a Test Assessment block, where the requirement is formalised using a MATLAB verification command such as *verify(margin_dB >= 6)*.

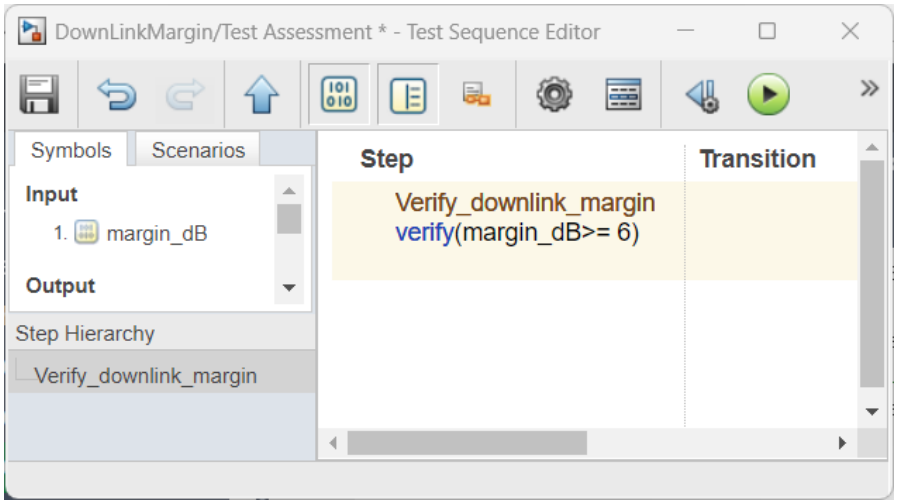


Figure 3.13: Test Assessment coding.

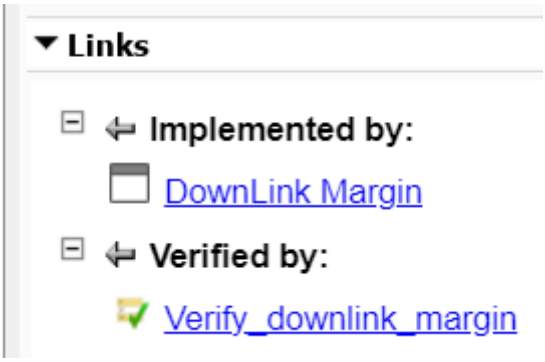


Figure 3.14: Requirement verification.

To ensure full traceability, each Test Assessment is linked to the corresponding requirement in the Requirements Manager. Initially, the Verification Status remains empty until the Test Case is executed in the Test Manager. During execution, the Test Manager runs the simulation, monitors the Test Assessment assertions, and records the results. Upon completion, the Verification Status column updates automatically to indicate Pass or Fail, providing auditable evidence of compliance. This approach ensures that deviations from expected link performance are immediately detectable, supporting a robust verification process critical for mission- or safety-critical satellite communication systems.

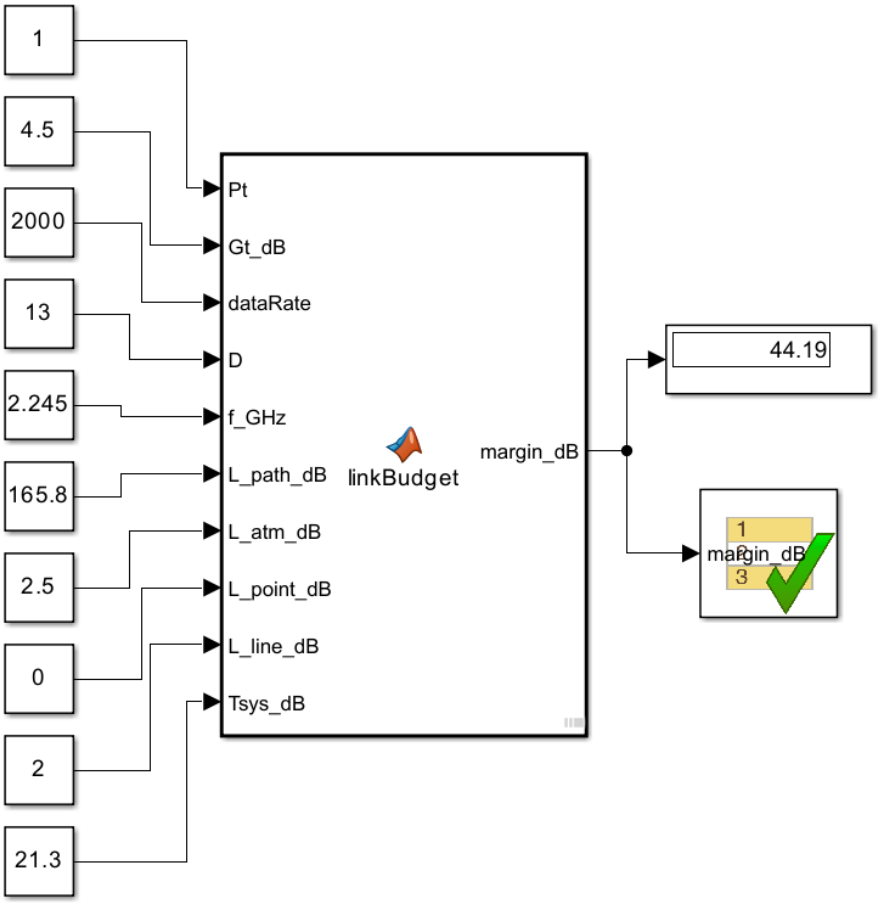


Figure 3.15: Component block view with Test Assessment linked to Requirement.

The methodology is also scalable and extensible. By leveraging Test Sequences and parameter sweeps, multiple operational conditions can be automatically evaluated, including variations in slant range, atmospheric attenuation, antenna misalignment, or system temperature. Each scenario's results, including *margin_dB* and the corresponding verification outcome, are captured in MATLAB for post-processing and in the Requirements Manager for formal traceability. This provides comprehensive coverage of the operational envelope and ensures compliance across nominal, degraded, and extreme conditions.

Chapter 4

Case Study I

The following example considers a 12U CubeSat, developed during academic activities, in order to further evaluate the capabilities of System Composer.
This study also served as an initial training case for the tool, enabling the assessment of its strengths and limitations through a less complex system, which does not include the additional variable of human presence, as further introduced in the case study presented in Chapter 5.

4.1 Stakeholders needs

In the realm of space exploration, a stakeholder refers to any individual, group, organisation, or entity that has a vested interest in, is affected by, or plays a role in the planning, execution, or outcome of a space mission. This encompasses far more than just the primary customer or funding body.
Stakeholders may be involved at various stages of the mission lifecycle, from initial concept development and technical design, through launch and operations, to data analysis and long-term impact assessment. Their engagement is essential to ensure that the mission is not only technically successful but also strategically relevant, socially beneficial, and aligned with broader scientific, economic, and policy goals. Stakeholders can influence mission objectives, resource allocation, risk management strategies, and public communication. Their expectations often drive innovation, collaboration, and accountability across national and international boundaries [31].

In this study, a comprehensive set of components has been developed in order to represent the principal stakeholders involved in the satellite project. Each component encapsulates one or more stakeholders, providing a structured representation of the stakeholder groupings, as outlined in Table 4.1. For clarity and consistency, each component has been assigned the stereotype "stakeholder", which System Composer compiles in the form *SatelliteProfile.Stakeholder*. This definition is not merely cosmetic: understanding the precise nomenclature employed by the software is fundamental for subsequent phases of the project, where variables must be retrieved and correctly mapped into the MATLAB Workspace. Within MATLAB, the software interprets these variables as *SatelliteProfile.Stakeholder.(name of the property)*, allowing them to be queried, manipulated, and incorporated into the broader modelling framework.
For the present study, a set of illustrative properties has been defined as follows:

- **Category** [enumeration]: from Apathetic, Defender, Promoter, or Latent;
- **Interest** [double]: from 1 to 10;
- **Power** [double]: from 1 to 10;
- **Needs** [string]: to indicate the specific stakeholder's need.

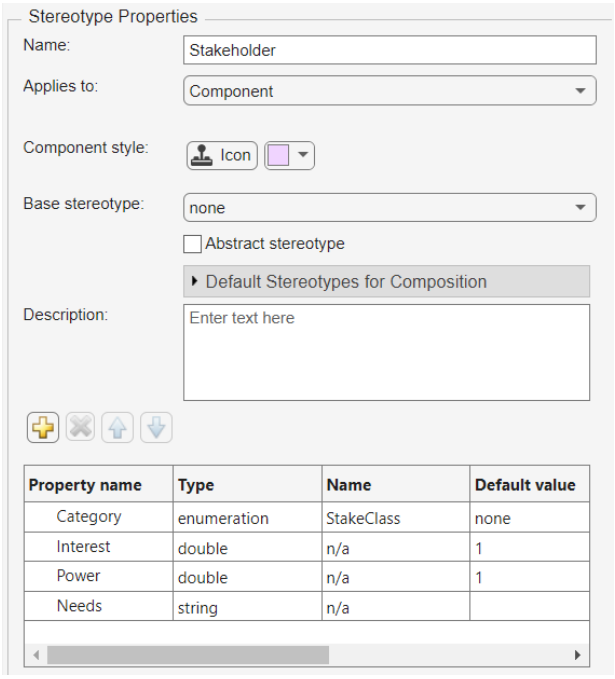


Figure 4.1: Stakeholders representation within System Composer.

The primary advantage of employing System Composer for stakeholder modelling lies in its ability to replace a static, tabular representation with a dynamic environment that affords greater flexibility during the design process. A conventional stakeholder table, although informative, tends to remain a fixed artefact with limited integration into subsequent analyses. By contrast, System Composer enables a living architecture, where stakeholders can be organised hierarchically, linked to specific system elements, and directly associated with trade-off considerations. Nevertheless, it is crucial to acknowledge the inherent complexity of stakeholders as entities: they are not reducible to fully deterministic models. Attempting to model every conceivable stakeholder interaction would be prohibitively expensive in terms of effort and computational overhead, and ultimately unnecessary for the practical objectives of system design. The central purpose of including stakeholders in the model is not to simulate their exact behaviour, but rather to highlight functional connections and identify areas of potential negotiation or conflict that could influence the mission.

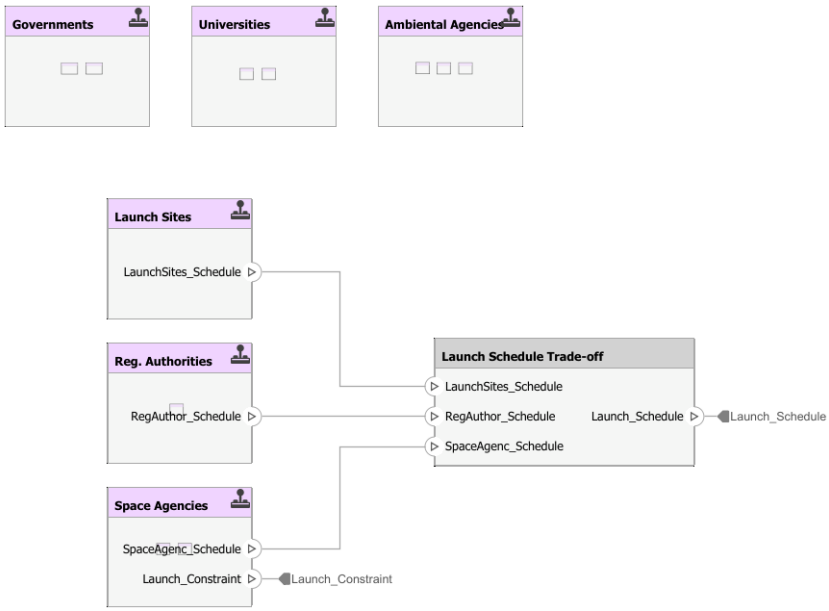


Figure 4.2: Stakeholders representation within System Composer.

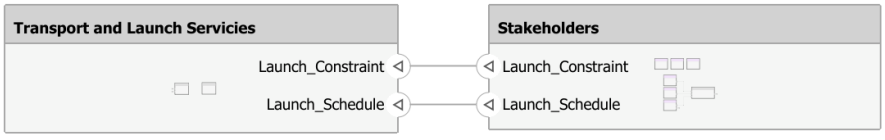


Figure 4.3: Stakeholders' outputs interaction with the architecture.

This distinction can be illustrated by examining Figure 4.2, which demonstrates how one might conceptualise a potential negotiation regarding the launch schedule of the satellite. At first glance, this might appear to imply the presence of an embedded computational routine within the "Launch Schedule Trade-off block". In reality, the figure functions instead as a high-level schematic, representing how the design environment may be employed to visualise potential interactions among stakeholders. The intention is illustrative rather than deterministic: it demonstrates that the architecture allows modellers to capture the existence of trade-offs without the need to encode the exact decision-making logic of each stakeholder. This ability to move between abstraction and structure underscores one of the central strengths of the tool.

Table 4.1: List of stakeholders.

Stakeholder family	**Stakeholder**
Governments	Italian municipalities
	European municipalities
Space agencies	ESA - UHI Programme
	Argotec
Regulatory agencies	ARPA
Universities	DIMEAS PoliTO, DIATI PoliMI
Ambiental agencies	Balfour-Beatty-Vinci
	Enel Green Power
	Trees for the cities

A particularly valuable feature of System Composer in this context is its support for nested architectures, which provide a means of organising stakeholders in a more granular and meaningful way. Rather than representing all stakeholders within a flat structure or a simple table, nested architectures permit the differentiation of various categories of stakeholders, for example distinguishing between regulatory bodies, funding agencies, scientific partners, and industrial contractors. This hierarchical organisation offers clearer traceability and improves the interpretability of the model. By embedding stakeholders within nested layers, the modeller gains the ability to illustrate relationships of authority, dependency, and collaboration more effectively than is possible with static documentation. This is especially important in projects such as satellite missions, where numerous parties contribute with differing objectives, levels of influence, and resources.

Beyond structural organisation, stakeholder analysis inevitably requires consideration of the relative weight of interests and powers that each actor brings to the

mission. Not all stakeholders exert influence to the same degree, nor do they share a uniform set of priorities.
While System Composer can be used to annotate and highlight these different priorities, it does not possess built-in mechanisms for assigning quantitative weights or for resolving conflicts between competing interests. Thus, while it can serve as an effective visual and organisational aid, the tool alone cannot capture the full complexity of stakeholder power dynamics.
This limitation becomes particularly evident when considering methodologies such as Quality Function Deployment (QFD). In stakeholder analysis, QFD is frequently employed to translate stakeholder needs into technical requirements, mapping qualitative priorities onto quantifiable design parameters. Unfortunately, System Composer does not currently provide native support for QFD frameworks. As a consequence, the modeller remains dependent on external tools (e.g. Microsoft Excel) to perform these translations. This dependency represents a structural limitation: while System Composer can host and display the relationships between stakeholders and system requirements, the rigorous quantification and prioritisation processes central to QFD cannot be fully replicated within its environment. The integration of System Composer with MATLAB does not overcome this gap, since MATLAB can access and manipulate the structural data but still lacks a dedicated QFD module. This highlights an unavoidable reliance on external tools for certain aspects of stakeholder analysis, a reliance which is both inconvenient and methodologically significant.

In conclusion, the adoption of System Composer for stakeholder representation within satellite mission design offers substantial benefits in terms of organisation, visualisation, and integration with the design environment. However, it should be approached with an awareness of its boundaries. The tool is best conceived not as a complete solution to stakeholder analysis, but as a complementary environment that supports and enhances conventional methods. Its strength lies in facilitating high-level reasoning about interactions and dependencies, while its weaknesses remind us of the persistent need for external frameworks to handle the quantitative and behavioural complexities of real stakeholders.

4.2 STM & TTM

For space mission design and systems engineering, the Science Traceability Matrix (STM) and Technology Traceability Matrix (TTM) are critical tools for ensuring alignment between mission objectives and implementation strategies.
The STM provides a structured framework that links high-level scientific goals to specific measurement requirements and instrument capabilities. It enables stakeholders to trace each scientific question through its associated observational needs, thereby ensuring that the payload and mission architecture are scientifically justified and technically feasible.
Conversely, the TTM focuses on the technological dimension, mapping system-level functions to enabling technologies and components. It supports the identification of critical technologies, their maturity levels (e.g., TRL), and their integration into the mission design. This matrix is essential for risk assessment, technology development planning, and design trade-offs [32]. Together, STM and TTM facilitate rigorous traceability, enabling transparent decision-making and robust verification processes throughout the mission lifecycle. Their use is standard practice in proposals to agencies such as NASA and ESA, where traceability is a prerequisite for funding and implementation.

The STM and the TTM are, by their very nature, artefacts that cannot be adequately recreated within the System Composer environment. Their primary purpose lies in the systematic collection, organisation, and preservation of scientific objectives and technical requirements. Attempting to reproduce such matrices through block-based modelling would inevitably distort and constrain their intended role, reducing their value as transparent and traceable reference instruments. For this reason, it has not been possible to construct an STM or TTM for this didactic example within System Composer. Instead, the traditional approach, based on comparative analysis and tabular schematisation, remains the most appropriate and conceptually faithful method for their development.

4.3 Mission design

4.3.1 Functional analysis

Functional analysis in space missions refers to a core process within systems engineering, in which mission objectives are systematically decomposed into logical and operational functions. Rather than focusing immediately on specific technological solutions, functional analysis asks what the system must do before addressing how it will be achieved. This methodological step ensures that the design remains objective-driven and not prematurely constrained by engineering choices.
The process typically begins with the identification of top-level scientific and operational objectives. These are broken down into primary and secondary functions, such as power generation and distribution, thermal regulation, communication, navigation, and data handling. Functional flow block diagrams and related modelling techniques are used to represent the logical interactions, inputs, outputs, and dependencies between functions.

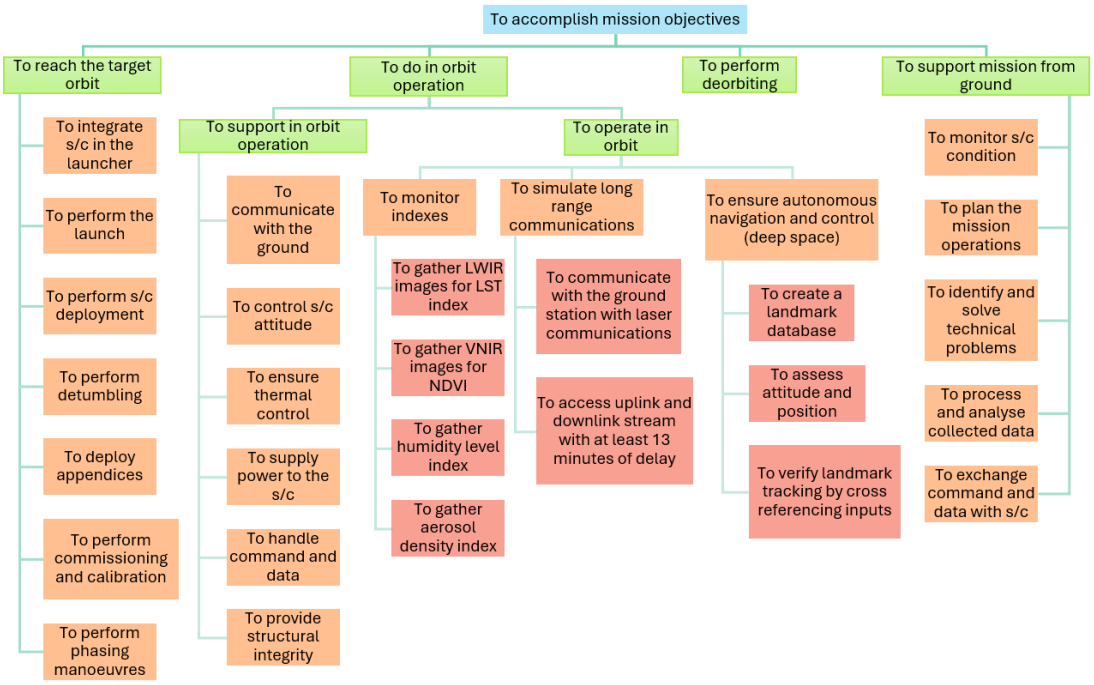


Figure 4.4: Example of a functional tree decomposition.

Once established, these functions provide a framework for the subsequent allocation to physical subsystems, such as payload instruments, spacecraft bus, or ground segment. Functional analysis is therefore crucial for maintaining consistency between mission goals and design solutions, for identifying redundancies, gaps, or

conflicts in requirements, and for enabling transparent traceability throughout the system lifecycle.[33]

A potential method for performing a functional analysis within the System Composer environment is to rely on the use of stereotypes. By assigning the stereotype *SatelliteProfile.Functional_analysis*, it becomes possible to associate each component with as many properties as the number of functions that the component is expected to fulfil. In practical terms, this means that the stereotype must be designed to contain at least as many properties as those required by the component with the greatest functional burden. Many components, however, typically perform only a single function, so the overhead is not always substantial.

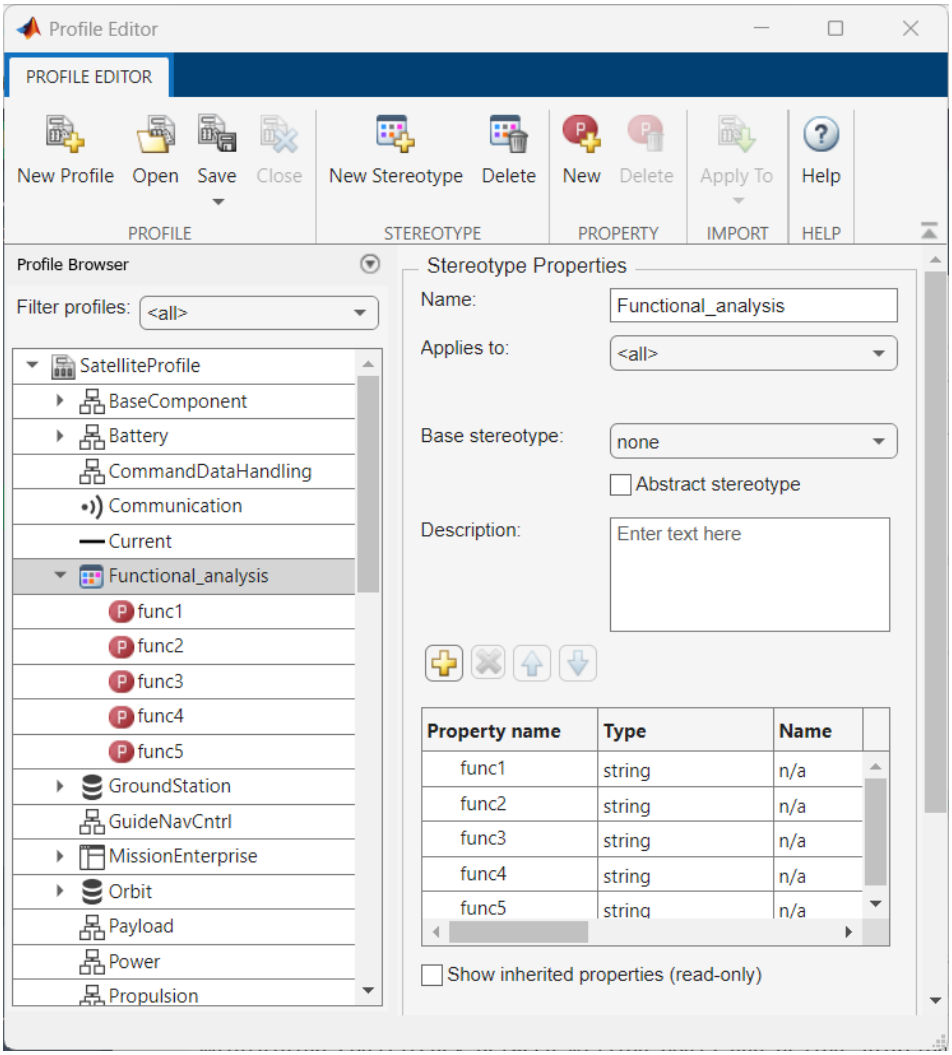


Figure 4.5: Functional analysis stereotype and properties.

While technically feasible, this approach is almost unorthodox. In the discipline of systems engineering, functional analysis is traditionally carried out using a hierarchical decomposition of functions, usually represented in the form of a tree. Such a structure progressively refines the system's objectives into major functions, sub-functions, and eventually specific tasks. This methodology not only clarifies the logical flow from high-level mission goals to detailed system behaviour but also provides transparency, traceability, and a global overview that can be reviewed and validated at each stage of the design process.
By contrast, the stereotype-based method acts more as a workaround than a systematic solution. Its main advantage lies in the ability to employ the architecture view within System Composer to filter and display functions directly in relation to components. This allows a degree of visibility into functional assignments. Nevertheless, the technique introduces important limitations. Firstly, filtering within the tool is case-sensitive, which creates a high risk of redundancy, duplication, or mismatches in the naming of functions. To mitigate this, it is strongly recommended to maintain an external and standardised list of functions that can serve as a reference throughout the analysis. Secondly, even with such measures, the approach cannot deliver a true top-level overview of the system's functional architecture. A further practical improvement is to allow functions to be associated not only with component blocks but with any modelling element — for instance Ports or Interfaces. Enabling such associations facilitates the generation of hypothetical logical flows during functional analysis, thereby supporting analysts in exploring how discrete functions might interact across the architecture without prematurely committing to a physical allocation.
The idea of using blocks to represent a hierarchical tree was considered but quickly dismissed, as external software tools offer greater flexibility and functionality compared to System Composer. Attempting to force the construction of such artefacts within the tool would result only in additional time losses and potential misunderstandings, due to the non-dynamic nature of tree hierarchy creation within the environment.

4.3.2 Mission requirements

Mission requirements represent the foundation of any space system development process, defining the objectives, performance targets, and operational constraints that guide subsequent design and verification activities.

While some requirements can be quantitatively verified through simulations or analytical assessments, others, particularly those defined at a high level, are generally verified by review or inspection. In this context, it is possible to formalise even these qualitative verifications within the model through the use of dedicated Test Assessments. These tests can incorporate direct user interaction, for example by enabling the user to manually modify a boolean variable to indicate whether a particular test has passed or failed. This method allows subjective or procedural verifications to be represented consistently within the digital environment, ensuring that all requirements, even those not directly linked to simulation data, remain traceable and verifiable in principle.
However, given that the didactic case study considered in this work extends only up to Phase 0/A of the mission design process, many mission requirements remain unverifiable at this stage. Their verification would occur much later in the development cycle, during system integration, operational validation, or even post-launch in the case of a single-launch mission. This limitation reflects the inherent nature of early design phases, where the focus lies primarily on feasibility assessment rather than complete validation. Consequently, the verification approach must be adaptive and forward-compatible, ensuring that all requirements can eventually be tested, confirmed, and traced as the mission progresses through its lifecycle.

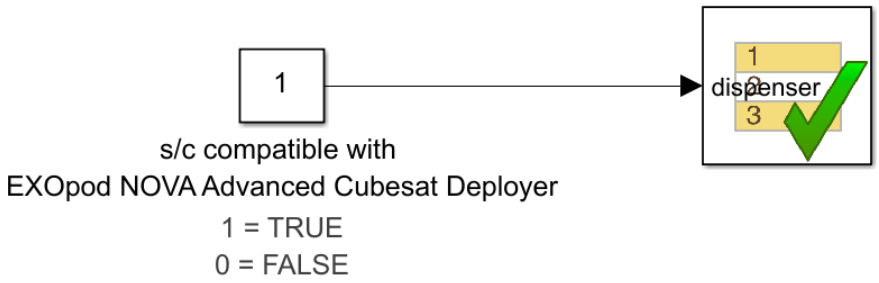


Figure 4.6: Example of a boolean verification.

4.4 Mission architecture

In the design of space missions, the mission architecture represents a fundamental element that provides a comprehensive and structured overview of the entire mission. It provides the principal components and their purposes, typically encompassing the mission subject, payload, space segment, geometry, ground segment, operations, launch segment, and communication architecture. By defining these elements, the mission architecture enables engineers and stakeholders to understand how each subsystem contributes to the fulfilment of mission objectives. It serves as both a conceptual and organisational tool, ensuring that the technical, operational, and programmatic aspects of the mission are coherently aligned from the earliest design phases. Despite its conceptual importance, the integration of a fully detailed mission architecture within certain modelling environments, such as System Composer, presents specific challenges.

The architecture cannot be directly embedded into the model without introducing significant abstraction or forcing the block-based representation beyond its intended use. While the block composition naturally offers a hierarchical view of the system, it does not inherently capture the full contextual relationships or high-level dependencies that characterise a mission architecture.

Attempting to reproduce the entire mission architecture through nested block diagrams would not only be redundant but could also reduce the clarity and readability of the system model.

The hierarchical block composition already provides a preliminary and practical representation of the architecture. Each subsystem, represented as a block with defined interfaces and parameters, contributes to a clear, layered understanding of the system's composition, although this representation is not as exhaustive as a dedicated architectural report or tabular summary.

Therefore, within System Composer, explicitly reconstructing a separate mission architecture is often unnecessary. The block-based design, with its nested structures, inherently provides a consistent overview of the system at various levels of abstraction. This approach ensures that designers can maintain a complete perspective of the mission while retaining the flexibility and traceability required for iterative development and verification activities.

4.5 Concept of Operation

As already explained in Section 3.2, the ConOps represents the central and critical element of mission design, providing the framework that links operational objectives to system functions and procedures. However, implementing the ConOps within System Composer presents several significant challenges. While it is possible to create a real sequence of interactions among system components with the *Sequence Diagram*, the absence of a well-defined syntax within the tool imposes important limitations. For instance, there is no native support for subdividing sequences into smaller subsequences, nor is there a facility for visual aids such as colour coding to distinguish different types of operations or phases. These shortcomings make the final artefact cumbersome, difficult to interpret, and highly prone to errors, inconsistencies, and redundancies.

Another key artefact within the ConOps is the Mission Timeline, which captures the chronological progression of mission events and activities. Within System Composer, the timeline can only be roughly approximated by using block components to represent discrete steps or events in a sequence. This approach, however, is inherently limited and constitutes a forced use of the software rather than a faithful representation of the temporal dimension. The resulting timeline lacks clarity, precision, and the intuitive readability that would normally be provided by purpose-built scheduling tools.
Consequently, while System Composer can support certain aspects of ConOps modelling, such as visualising component interactions or functional allocations, it cannot fully replace specialised software for the detailed development of operational sequences or mission timelines. Users must therefore accept significant compromises in terms of accuracy, completeness, and usability when attempting to implement these artefacts solely within the tool. The limitations underscore the importance of complementing System Composer with external tools or methods to ensure that the ConOps and associated timelines are both comprehensive and reliable.

4.5.1 Operative Modes and State diagram

A potential approach to dynamically visualising the operative modes of the mission is to employ a State diagram, as seen in Figure 4.9, via a series of the Component blocks, complemented by a tailored stereotype named *SatelliteProfile.operativeModes*. This stereotype can incorporate as Properties all relevant components and subsystems selected for investigation in subsequent analyses, for instance the power budget. By linking these Properties to the operative modes, the representation facilitates both clarity and extensibility of the system model. Furthermore, through the dedicated function *fnc_modes().m*[1], it becomes possible to generate an output matrix capturing the relationships between modes and subsystems, which can then serve as a structured input for future static analysis.

	LWIR	VNIR	PCDU	comSys	OBC	starTracker	sunSensors	reactionWheels	IMU	magnetorquers	GNSS	PCU	thrusters
Dormant	0	0	0	0	0	0	0	0	0	0	0	0	0
Detumbling	0	0	0	0	1	0	0	0	1	1	0	0	0
Commissioning	0	0	1	1	1	1	1	1	1	0	0	0	0
OrbitInsertion	0	0	1	1	1	1	1	1	1	0	1	2	2
Nominal	0.5	0.5	1	0.5	1	1	1	1	1	0	1	0	0
Manouvering	0.5	0.5	1	0.5	1	1	1	1	1	1	1	2	2
PayModeGround	0.5	1	1	0.5	1	1	1	1	1	0	1	0	0
PayModeCity	1	1	1	0.5	1	1	1	1	1	0	1	0	0
Transmission	0.5	0.5	1	1	1	1	1	1	1	0	1	0	0
SafeMode	0	0	1	1	1	0	0	1	0	0	0	0	0
Disposal	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.7: *fnc_modes('fullModes')* first output.

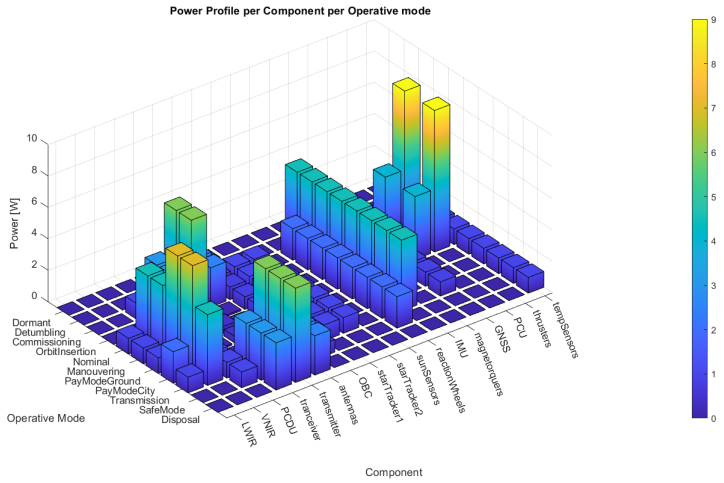


Figure 4.8: *fnc_modes('fullModes')* second output.

[1]See Appendix A for details on fnc_modes().m

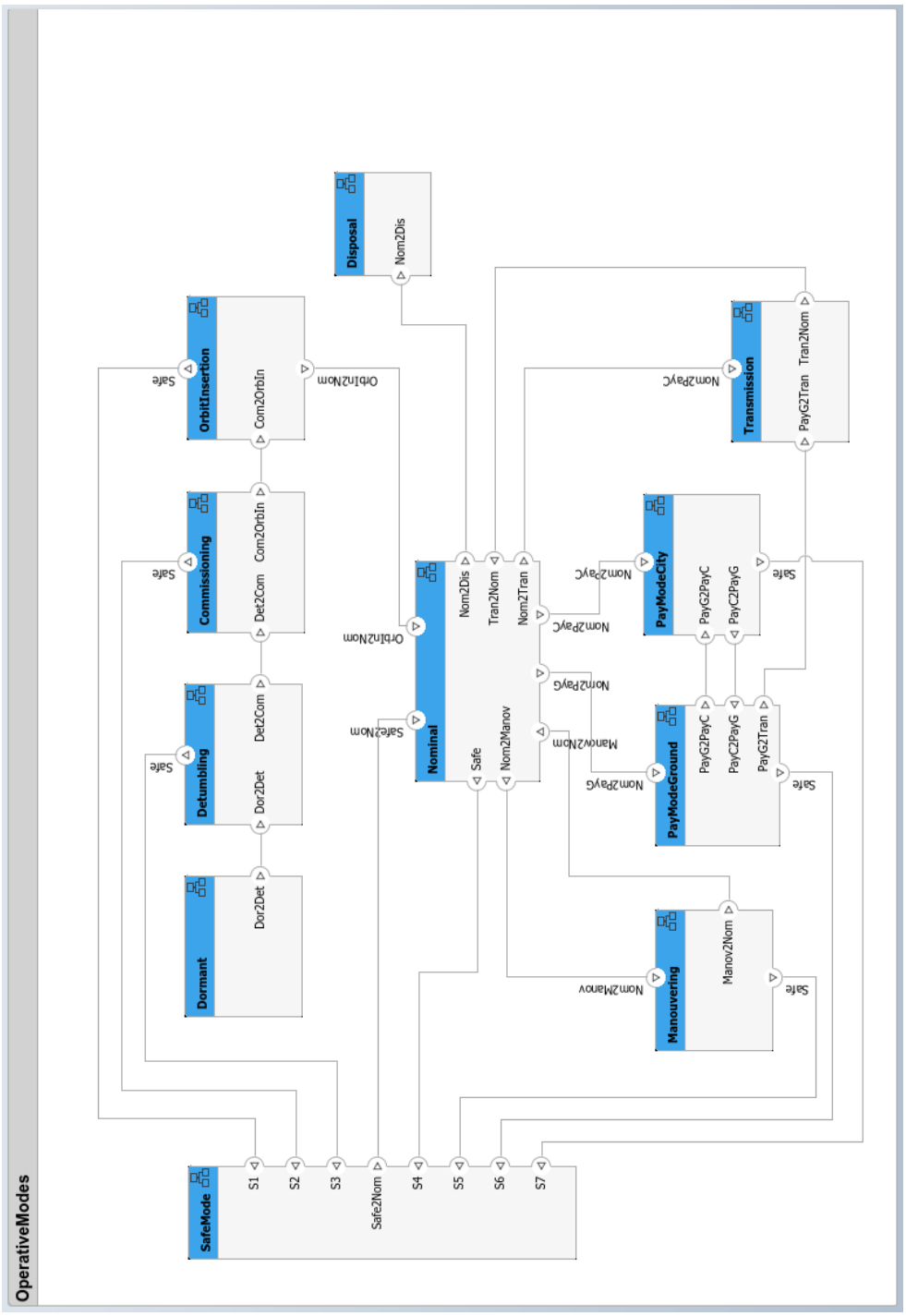


Figure 4.9: State diagram.

4.6 Mission analysis

Within mission analysis, several parameters are of fundamental importance, including the selection of the spacecraft trajectory, satellite coverage, the computation of the delta-V budget, and the estimation of the disposal time required at the end of the operational phase. Because the objective is to reproduce and analyse a realistic space environment, it is practically impossible to proceed without the aid of external software tools. During the early design stages, such tools provide configurable and pre-defined parameters that allow for a clear and comprehensive representation of the mission scenario. For example, the STK Propagator is commonly employed to assess orbital coverage, propagation, and perturbations, while the OSCAR tool is often used to model and evaluate end-of-life disposal strategies and orbital decay processes. However, when the principal goal is to define or optimise the trajectory, the entire system must first be modelled in sufficient detail. Only by doing so is it possible to solve the governing equations of orbital dynamics, such as those describing the two-body problem, within a physically meaningful framework. This dependency highlights a significant limitation: parameters such as trajectory, coverage, or disposal cannot realistically be fixed in the earliest phases of design without an already developed system architecture.

4.7 Risk assessment

Risk assessment within System Composer remains a challenging task, primarily because the tool is not designed to natively manage the artefacts typically used in risk management.
Standard practices in this field rely on the systematic use of risk registers and risk matrices, which enable both the categorisation and prioritisation of risks across the system. System Composer, however, lacks the mechanisms to create and dynamically update these artefacts, making it difficult to integrate risk assessment directly within the modelling environment.
A possible workaround consists of associating risks with the most critical components through the use of tailored stereotypes, offering some advantages, such as maintaining a degree of traceability between architectural elements and the risks that affect them, but presenting several significant limitations. First, the risks encoded in the model cannot be effectively processed or evaluated without the support of external tools. Second, the manual entry of such information increases the probability of inconsistencies and errors. Finally, embedding excessive risk-related annotations directly into the system architecture can "pollute" the model with superfluous details. For these reasons, comprehensive risk assessment should be conducted with specialised external solutions.

4.8 Cost assessment

The cost assessment is the systematic process of estimating, analysing, and evaluating the total costs associated with a project, product, or system throughout its life cycle. It aims to provide a clear understanding of how financial resources will be allocated, identifying cost drivers, uncertainties, and potential risks, supporting decision-making by allowing trade-offs between performance, schedule, and budget. Within System Composer, this can be achieved through a simple static analysis using a tailored MATLAB function, *fnc_budget().m*[2], where *type* is set to 'cost'.

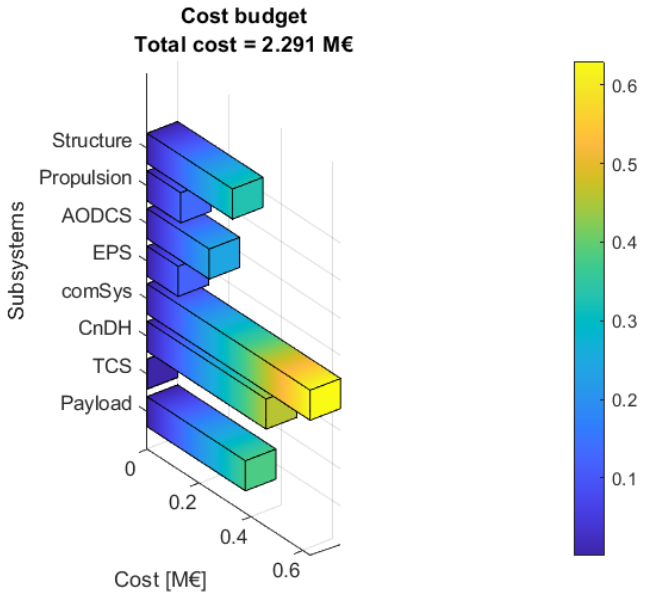


Figure 4.10: Cost budget.

[2]See Appendix A for details on *fnc_budget()*.m

4.9 System architecture

Beyond the traditional nested system architecture, a set of stereotypes has been introduced to enhance the descriptive capability of the model, enabling a more detailed and structured representation of the individual components and their interrelations within the overall system framework.

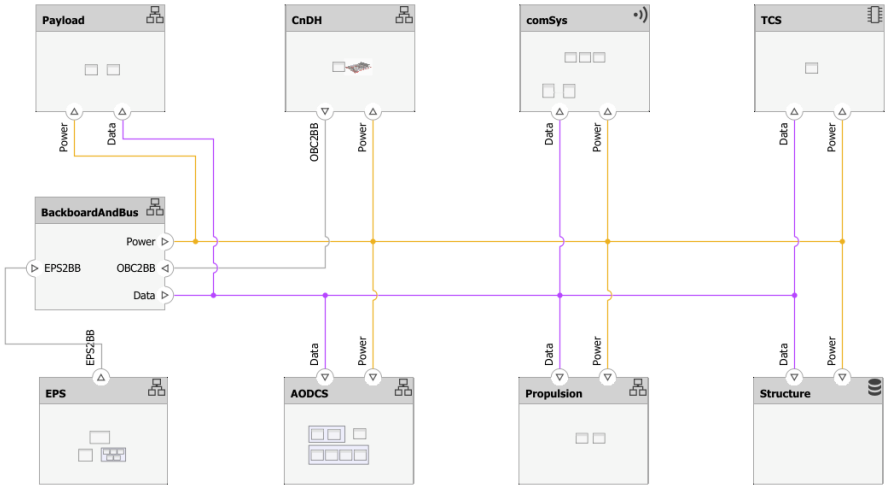


Figure 4.11: Native system architecture within System Composer.

For this didactic case, a high-level architecture has been developed with the aim of offering a comprehensive overview of the principal power and data interactions within the system. In addition to outlining the overall structure of these interactions, the interface also establishes a coherent colour scheme, which serves as a valuable visual aid.

Via the *Profile Editor*, the following following stereotypes fill the system:

Table 4.2: Battery stereotype properties.

Property	**Units**	**Scope**
parallelCells	double	Number of in parallel cells
seriesCells	double	Number of in series cells
capacity	Wh	Maximum battery capacity
producedPower	W	Maximum output power

Base stereotype: *SatelliteProfile.BaseComponent*
System Composer directory: *SatelliteProfile.Battery*

Table 4.3: Solar array stereotype properties.

Property	Units	Scope
parallelCells	double	Number of in parallel cells
seriesCells	double	Number of in series cells
producedPower	W	Maximum output power

Base stereotype: *SatelliteProfile.BaseComponent*
System Composer directory: *SatelliteProfile.SolarArray*

Table 4.4: BaseComponent stereotype properties.

Property	Units	Scope
compID	str	Component's/system's reference ID
mass	kg	Define component's/system's mass
cost	M€	Define component's/system's cost
powerConsStb	W	Power consumption while in *Stand-by* mode
powerConsOn	W	Power consumption while in *On* mode
powerConsPeak	W	Power consumption while in *Peak* mode
compName	str	Component's/system's name
manifacturer	str	Component's/system's manifacturer
volume	str	System external volume

System Composer directory: *SatelliteProfile.BaseComponent*

Table 4.5: Receiver stereotype properties.

Property	Units	Scope
gain	dB	Receiver antenna gain
lossCable	dB	Signal losses due to cable length
lossConnector	dB	Signal losses due to connectors
lossPoint	dB	Signal losses due to pointing mismatch
noiseTemp	K	System Noise Temperature

Base stereotype: *SatelliteProfile.BaseComponent*
System Composer directory: *SatelliteProfile.Receiver*

Table 4.6: Transmitter stereotype properties.

Property	**Units**	**Scope**
power	W	Transmitter antenna power
gain	dB	Transmitter antenna gain
lossCable	dB	Signal losses due to cable length
lossConnector	dB	Signal losses due to connectors
lossPoint	dB	Signal losses due to pointing mismatch
lossMisc	dB	Other signal losses
freq	Hz	Transmitted signal frequency
dataRate	kbit/s	Transmitter data rate
reqBER	float	Required Bit Error Ratio
modulation	str	Modulation type name
sysMargin	dB	Margin applied to the transmitted signal

Base stereotype: *SatelliteProfile.BaseComponent*
System Composer directory: *SatelliteProfile.Transmitter*

Table 4.7: GroundStation stereotype properties.

Property	**Units**	**Scope**
Lat	deg	GS latitude
Lon	deg	GS longitude
altitude	m	GS altitude
minElevation	W	Maximum output power
powerTx	W	GS transmitter power
lineLossTx	dB	GS transmitter signal losses due to line length
lossConnectorTx	dB	GS transmitter signal losses due to connectors
gainTx	dB	GS transmitter gain
lossPointTx	dB	Gs signal losses due to pointing mismatch
freq	Hz	GS transmitted signal frequency
dataRateTx	kbit/s	GS transmitter data rate
reqBer	float	Required Bit Error Ratio
modulation	str	Modulation type name
sysMargin	dB	Margin applied to the transmitted signal
diameterRx	m	GS receiver antenna diameter
LNAGain	dB	GS receiver Low Noise Amplifier gain
noiseTempRx	K	GS receiver System Noise Temperature
lossCableRx	dB	GS receiver signal losses due to cables length
lossConnectorRx	dB	GS receiver signal losses due to connectors

System Composer directory: *SatelliteProfile.GroundStation*

4.9.1 Product tree

A product tree is a hierarchical decomposition of all components, subsystems, and assemblies that constitute a satellite, structured to represent the product's complete architecture, defining the relationship between high-level system elements.
This structured breakdown facilitates configuration management, traceability of requirements, and efficient coordination between engineering teams. Ultimately, the product tree serves as a foundational tool in systems engineering, ensuring that every element of the satellite is accounted for throughout its design, manufacturing, integration, and verification phases.

With the *Architecture View*, it is possible to automatically generate a product tree within the system. In fact, by applying a simple filter such as *Select Components Where Stereotype is SatelliteProfile.BaseComponent*.

When the number of elements increases, the views can easily become cluttered. To address this, a focus view can be applied to provide a clearer representation of a single branch of the tree, as shown in Figure 4.13.

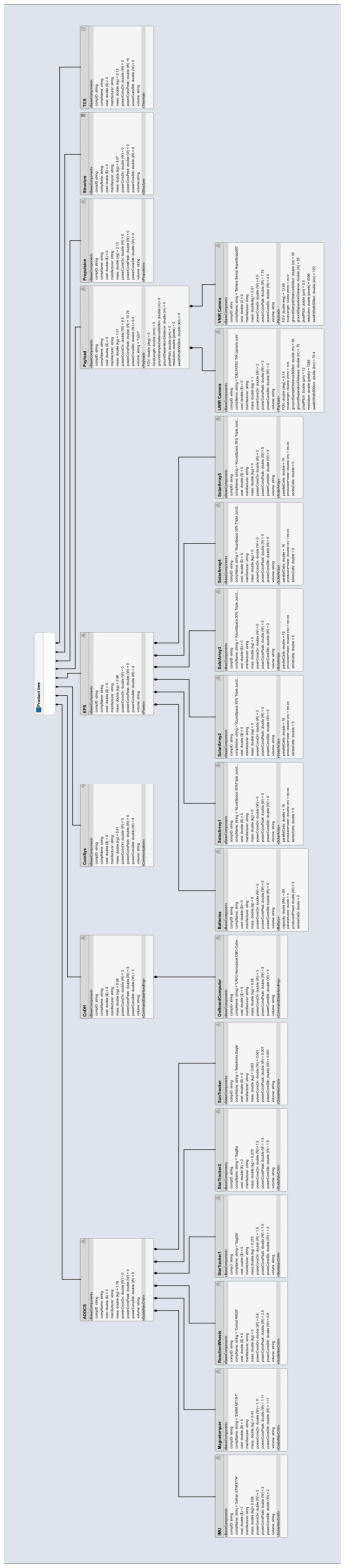


Figure 4.12: System Composer product tree.

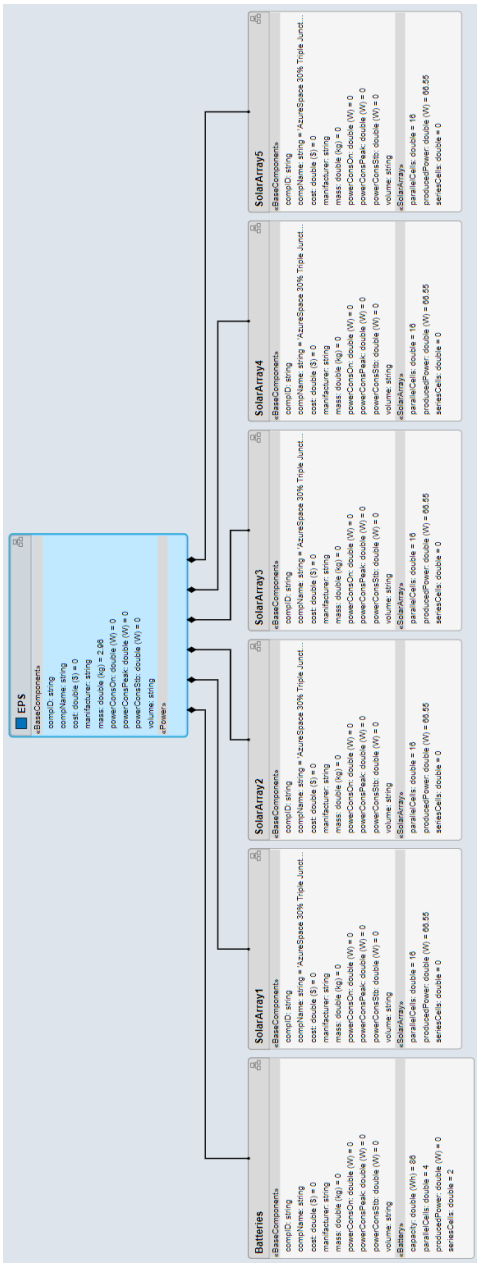


Figure 4.13: System Composer product tree focused branch.

4.10 System budgets

4.10.1 Mass budget

In spacecraft mission design, conducting a detailed mass budget is a fundamental step to ensure the feasibility, safety, and cost-effectiveness of the mission. It provides a comprehensive accounting of all components, subsystems, and payloads, allowing engineers to verify that the total mass remains within the launch vehicle's capacity and that appropriate margins are maintained for contingencies, since even small deviations in mass can significantly affect propulsion requirements, orbital parameters, and mission duration.

Within System Composer, the mass of each subsystem or component can be conveniently assigned through the use of Properties (e.g. SatelliteProfile.Payload.mass), enabling a static and simple assessment of the spacecraft's structural and functional elements. However, while this manual attribution is effective for initial design stages, it becomes inefficient when dealing with complex architectures involving numerous interconnected components. To address this limitation, a tailored function was developed to systematically access the architecture, read relevant parameters, and assign to predefined variables the key data required for analysis. This approach facilitates consistent and automated data extraction across multiple components, improving both accuracy and traceability. For instance, the function *fnc_budget().m*[3] demonstrates how, starting from a specified list of components and their associated stereotypes, it is possible to extract individual mass values and compute the total system mass.

The verification of the mass budget within the system is consistently performed through a Test Assessment implemented inside a component containing a Simulink Behaviour. In this configuration, the required variable is provided via a Constant block, ensuring controlled and reproducible input during each verification cycle. Particular care must be taken to avoid naming the block's behaviour with the same identifier as the function used for mass calculation, as this may cause the system to misinterpret the variable assignment. Maintaining distinct naming conventions prevents conflicts during model execution and ensures reliable and unambiguous evaluation of the mass budget results.

[3]See Appendix A for details on *fnc_budget()*.m

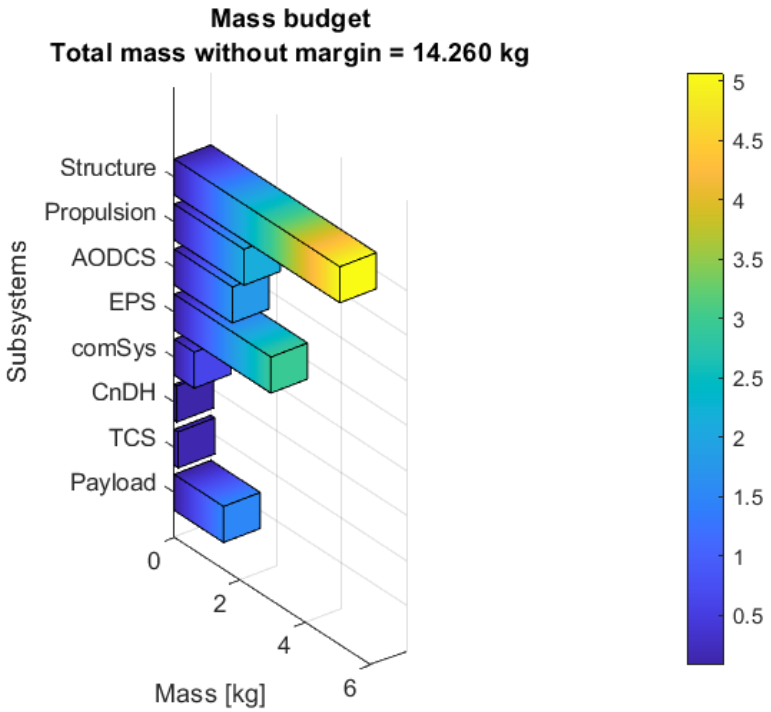
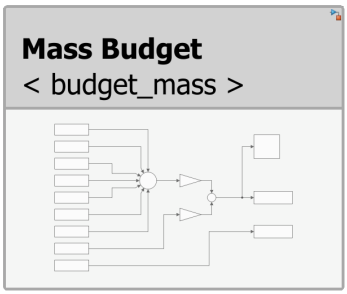
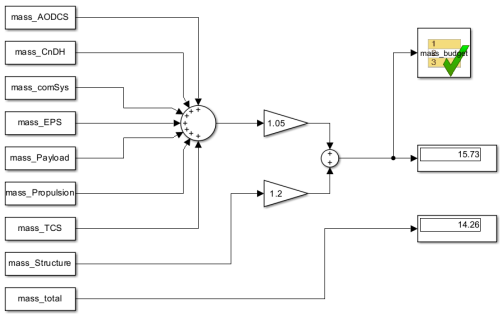


Figure 4.14: Mass budget.



(a) Component block with Simulink Behaviour.



(b) Simulink *Constant*, *Display* and *Test assessment* blocks.

Figure 4.15: Mass budget verification within System Composer.

4.10.2 Power budget

A power budget is a fundamental component of any spacecraft mission design, serving as a systematic assessment of the electrical power generation, distribution, and consumption throughout the spacecraft's operational lifetime. Its primary purpose is to ensure that the available energy, typically provided by solar arrays and stored in batteries, satisfies the demands of all subsystems under a variety of mission modes.
By quantifying both nominal and peak power requirements, the power budget allows engineers to verify design feasibility, identify critical operational constraints, and optimise the allocation of energy resources. Moreover, it supports the definition of safe operating margins, ensuring continuous functionality even in degraded or eclipse conditions. Thus, the power budget plays a central role in balancing performance, reliability, and mass efficiency within the overall spacecraft architecture.

For this case study, a series of dedicated functions were provided to facilitate navigation through the system architecture and to retrieve the necessary data for power-related analysis. For instance, the creation of a well-defined power budget requires an accurate understanding of the power demand associated with each spacecraft component. This task is accomplished through the function *fnc_powerProfile().m*[4], which generates both a matrix and a vector automatically employed to produce the "Power Profile per Component" and the "Power Profile per Subsystem", as shown in Figure 4.16 and Figure 4.17.

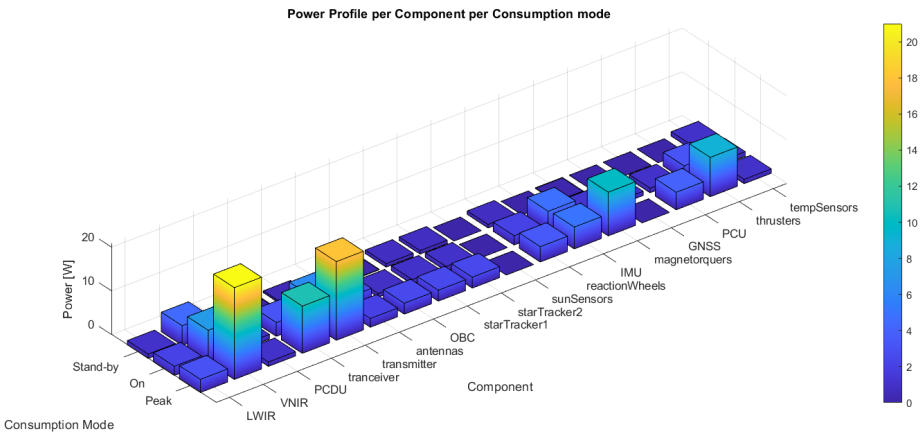


Figure 4.16: Power Profile per Component.

[4]See Appendix A for details on *fnc_powerProfile()*.m

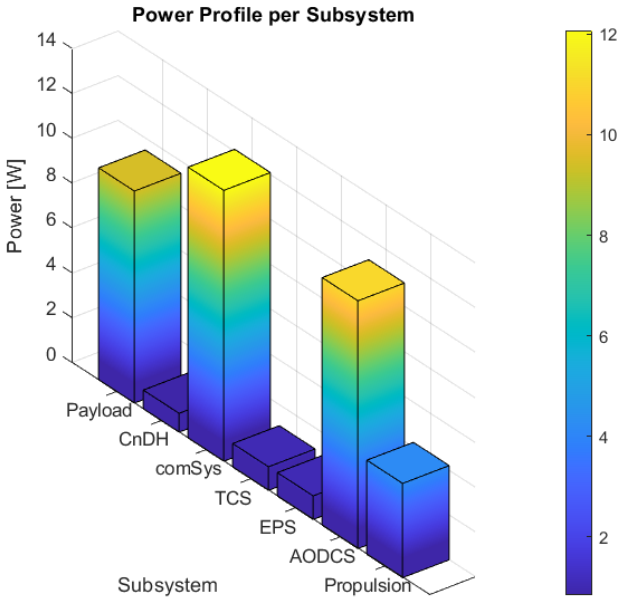


Figure 4.17: Power Profile per Subsystem.

Continuing this procedure, the implementation of the function *fnc_modes.m*[5] within the script *powerConsOrbit.m*[6] enables the generation of a "Power Consumption per Mode" plot, shown in Figure 4.18. Additionally, it allows for the definition and analysis of mode combinations in a duty cycle configuration. These combinations are visualised through a bar chart, depicted in Figure 4.19. The resulting Power Budget, taking in account the analysis the second and most articulated of the various combinations: Transmission - Nominal - Manouvering, is presented in Figure 4.20.

[5]See Appendix A for details on *fnc_modes()*.m

[6]See Appendix B for details on *powerConsOrbit.m*

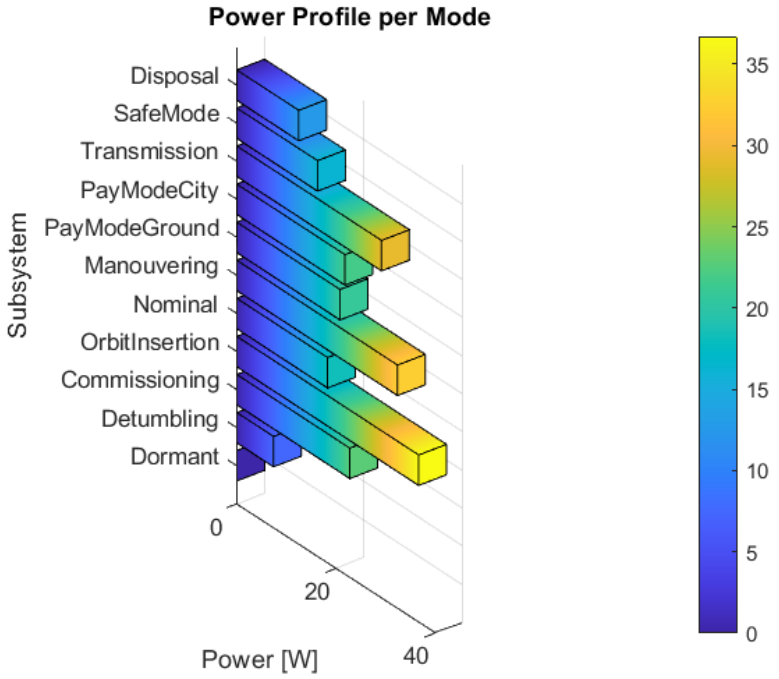


Figure 4.18: Power Profile per Mode.

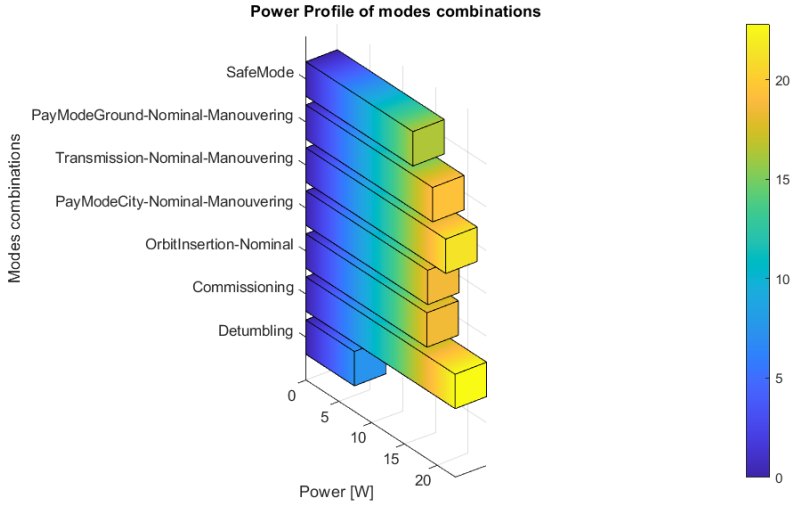


Figure 4.19: Power Profile of Mode Combinations.

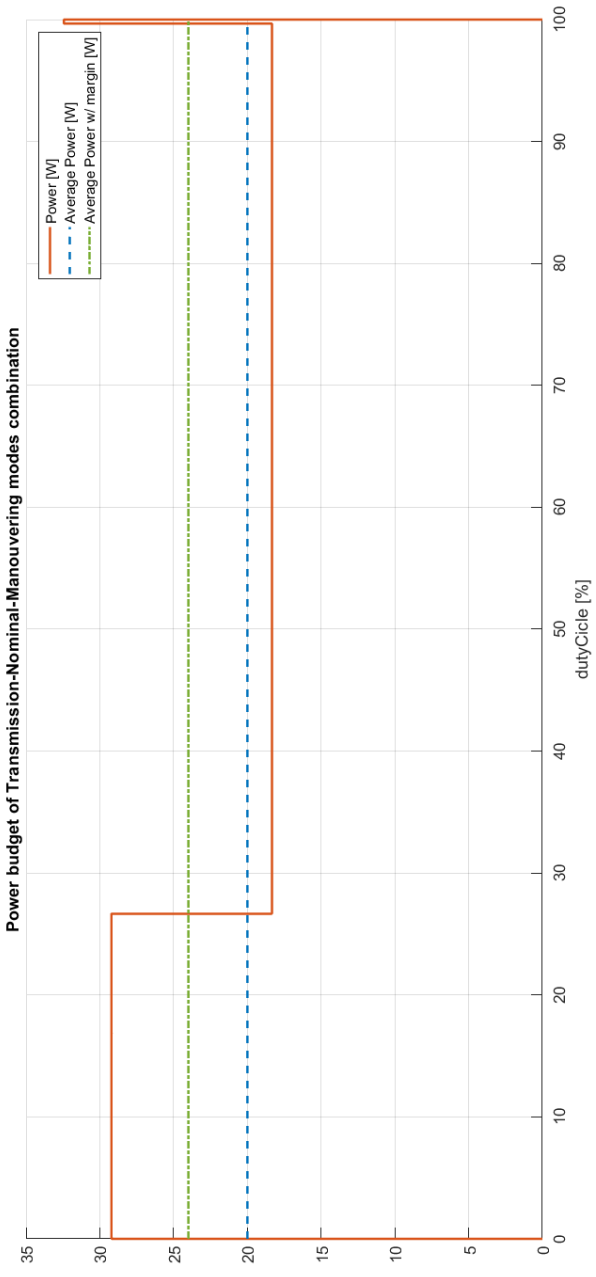


Figure 4.20: Power Budget of Transmission - Nominal - Manoeuvring modes combination.

Chapter 5

Case study II

In this chapter, building on the practical experience gained through the CubeSat case study (see Chapter 4), which demonstrated the feasibility of high-level satellite implementation when appropriate design considerations are applied, the focus shifts to analysing how human needs can be incorporated into the design process. This is achieved by outlining the reasoning and design approach underlying the study of the ISS Environmental Control and Life Support System (ECLSS).
The human being presents a wide and diverse set of needs which, when accounted for within a multi-subsystem architecture, significantly increase the overall system complexity. Incorporating human requirements into the design process leads to an exponential growth in the number of elements, components, interfaces, and mass and energy flows that must be identified, modelled, and managed.
This level of detail is essential in order to produce a system representation that is sufficiently realistic to support meaningful analysis and simulation. Consequently, particular attention is devoted to structuring and organising these elements in a coherent manner, ensuring that the resulting model remains both comprehensive and tractable from an engineering and MBSE perspective.

5.1 Human-Centred Engineering Principles

The entire mission team must incorporate human factors as early as possible during mission analysis, concept development, and system design. Too frequently, programmes address the human element only after other critical design decisions have been made, which can give rise to mission or operational problems that might otherwise have been avoided. In the design of systems for human operators, physiological individuality is also crucial: each person exhibits differences arising from genetic makeup, ethnic and cultural background, gender, and prior experiences shaped by the environment or personal habits [34].

5.2 Physiological requirements

Leaving the Earth's surface to reach space requires humans to adapt to an environment that is fundamentally different, with conditions unlike those on Earth. The primary factors considered in this study are: habitat atmosphere and ventilation, radiation, acceleration, noise, vibrations, metabolism, microgravity, and psychological effects [35]. The context of a space mission environment is summarized in Figure 5.1.

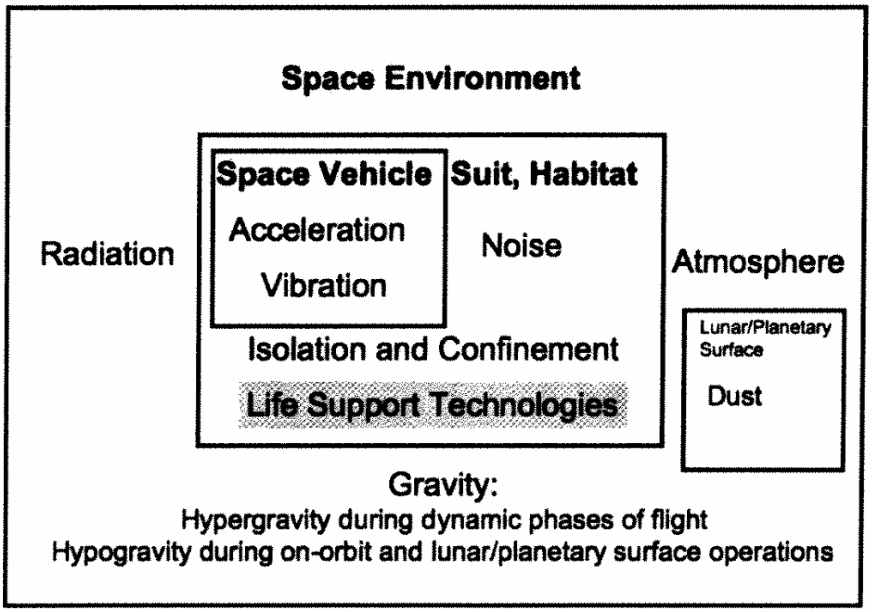


Figure 5.1: Overview of environmental factors [36].

Habitat atmosphere and ventilation

By relying on pressurized structures, it's fundamental to maintain an appropriate atmospheric and temperature compositions.
The atmosphere can be defined via the partial pressures (p) of its core components:

$$P_{total} = pO_2 + pN_2 + pH_2O + pCO_2 + pX_1 + pX_2 + \cdots + pX_n$$

where P_{total} is the total barometric pressure, N_2 is the inert gas, and pX_n represent the partial pressure of components or contaminants. The trade-off involves P_{total} and pO_2, with pN_2 as diluent. For a dry-air profile, the simplest estimate for nitrogen is $pN_2 = P_{total} - PO_2$. To support human health, the partial pressure of O_2 must remain around 21.4 kPa at sea level. In fact, also excessively high O_2 partial pressure can be harmful, which can lead to conditions such as lung irritation and oxygen toxicity [35, 34].

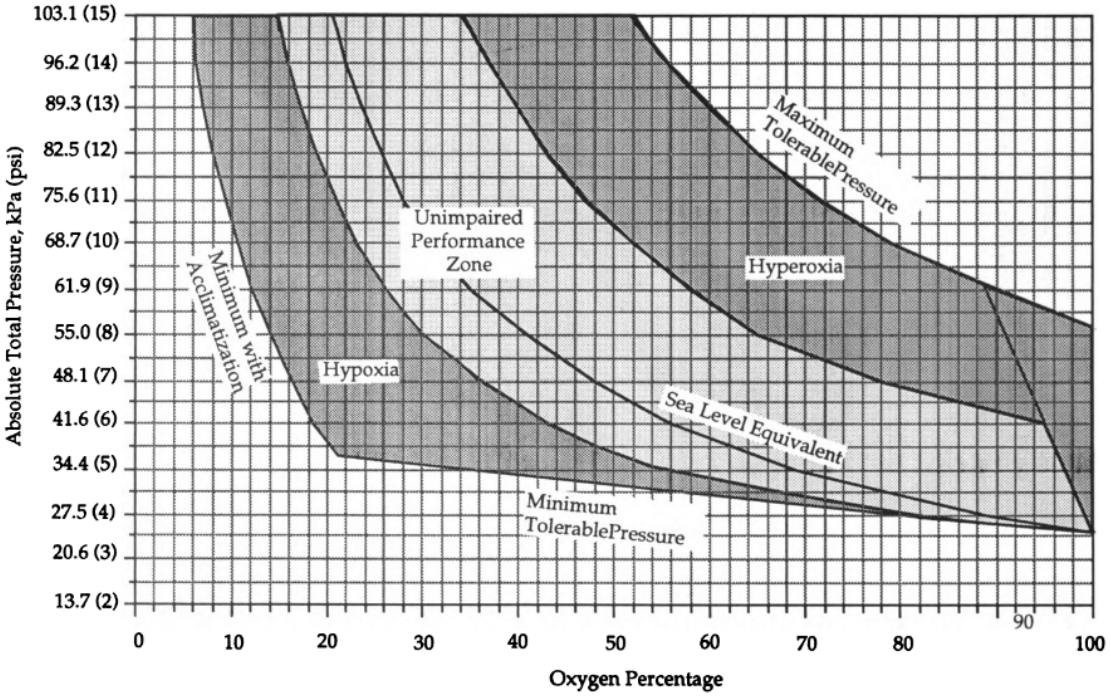


Figure 5.2: Physiological effects of oxygen concentrations [37].

As shown in Figure 5.2, its critical to select and maintain total pressure and oxygen partial pressure within the light-shaded area to meet the human physiological requirements, which extends from a total pressure of 103.1 kPa at 21% oxygen to a total pressure of 25 kPa at 100% oxygen (based on the sea level equivalent curve in the diagram) [35].

There are several advantages and disadvantages associated with using either a sea-level atmosphere or a reduced-pressure atmosphere within the spacecraft cabin. Consequently, in addition to meeting human physiological requirements, the following factors must be taken into account when determining the appropriate total cabin pressure:

- Maintaining a total sea-level pressure results in greater atmospheric losses due to leakage, requires additional structural mass to withstand the higher pressure differential between the interior and exterior of the habitat, and may complicate EVA preparations.
- Conversely, lowering the total cabin pressure below sea level makes voice communication over distances more difficult, increases the risk of fire due to a higher oxygen concentration, alters the environmental conditions for on-board experiments relative to those on Earth, enhances material off-gassing, and reduces the efficiency of heat transfer [34].

In a finite atmosphere, it is essential to control temperature and humidity to provide a comfortable environment for the crew reducing the risk of condensation on electronic components. Moisture accumulation in an area can create good condition for microbial growth and spread, due the restricted convection ventilation and reduced gravity.
A desirable range of Relative Humidity (RH) is between 25%-70%; if too low,crew members may experience drying of the nose, throat and other respiratory issues, while if the humidity is too high, a more common problem in space habitats, perspiration doesn't cool the crew properly, condensation on surfaces may occur and micro-organisms may grow more rapidly [37].
Temperature requirements have generally been the same for all space habitats, ranging from 18.3 °C to 26.7 °C.At high temperatures, a low RH is required to facilitate evaporative cooling. On the other hand, at low temperatures, a high RH is preferred to minimize heat loss through evaporation and maintain comfortable conditions [35].

The *comfort box*, illustrated in Figure 8, represents the optimal range of environmental conditions required to maintain a stable and comfortable atmosphere suitable for sustaining human life.

In addition, poor airflow within the environmental control system can lead to several operational and physiological issues. First, due to the variety of crew activities, "stale" air and CO_2 can accumulate locally, creating zones of reduced air quality. Second, insufficient ventilation results in the build-up of airborne contaminants that would otherwise be captured and removed by the filtration and decontamination systems. Third, inadequate air circulation slows the evaporation

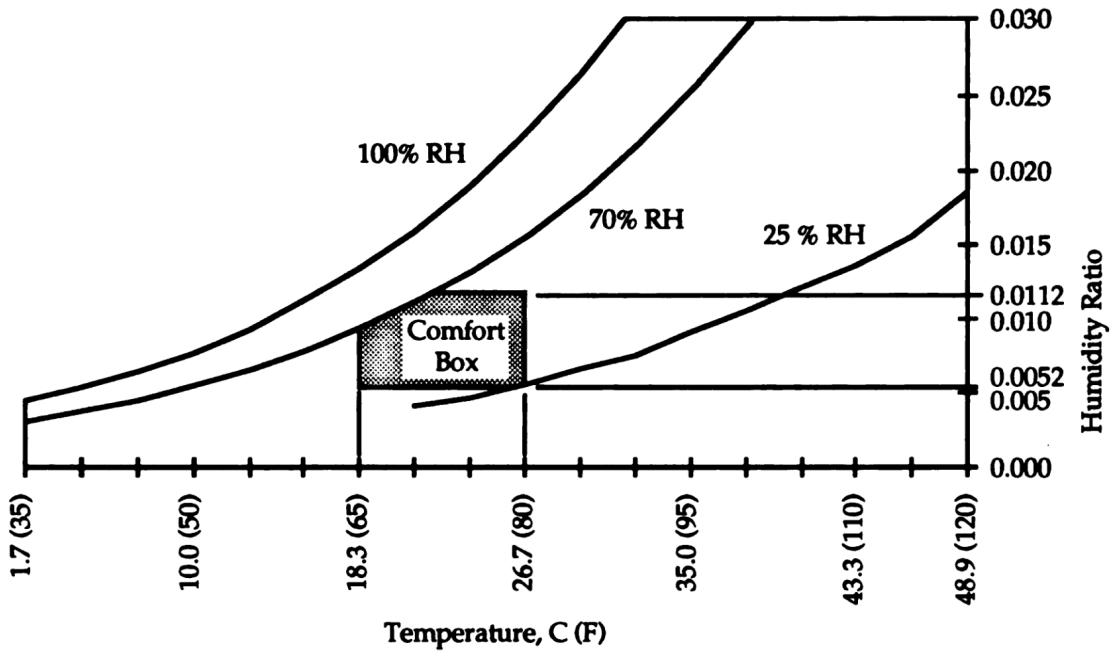


Figure 5.3: Temperature and RH ranges [37].

of sweat, increasing humidity and thermal discomfort, which may reduce crew performance and well-being.
On the International Space Station (ISS), during a 90-day mission, the general airflow velocity is maintained between 0.08 m/s and 0.20 m/s. The system also provides enhanced ventilation in "dead spots," where airflow can reach 0.42 m/s or higher, particularly in crew exercise areas or regions where metabolic output is increased.

Radiations

As humans remain in space for extended periods, they require reliable protection against space radiation, particularly from charged particles. Three main types can be identified:

- **Solar Particle Events (SPEs):** energetic protons originating from solar coronal mass ejections.
- **Galactic Cosmic Rays (GCRs):** high-energy atomic nuclei originating outside the Solar System.
- **Van Allen radiation belts:** regions of energetic particles trapped along Earth's magnetic field lines [35].

Regardless of mission type, the crew requires specific protective strategies, especially during intense solar events. The structural elements of the spacecraft can provide partial shielding, and lightweight materials such as polyethylene, as well as supplies like water or propellant, can be effectively used as supplementary barriers. For instance, a "safe haven" equivalent to an aluminium sphere of 2 m diameter with a 7.5 cm thick wall (20 g/cm^2) would weigh approximately 2.5 tons [34].

Acceleration

While planning a space mission, engineers must carefully assess the acceleration environment with respect to human tolerance. In particular, attention must be given to linear acceleration, rotational acceleration, and impact.
When analysing these effects, it is essential to consider the direction of the acting force relative to the human body. Assuming an upright seated posture, the x-axis runs from chest to back, the y-axis from side to side, and the z-axis from head to seat. The human body shows the highest tolerance along the $\pm g_x$ axis; therefore, most high-g systems are designed to align the crew along this direction.
According to NASA-STD-3000, during an emergency re-entry the acceleration should not exceed $\pm 4, g_x$, $\pm 1, g_y$, or $\pm 0.5, g_z$.
For rotational acceleration, untrained individuals can generally tolerate up to 6 rpm in any direction, although training can substantially increase this limit.
Human tolerance to impact acceleration depends not only on the peak g-forces experienced but also on the rate at which these forces are applied to reach the peak value [34].

Noise

Typical noise sources include motors, fans, pumps, valves, regulators, transformers, oscillators, and other equipment operating near thruster firings. Noise must be properly damped to preserve hearing, enhance crew comfort and life quality (including sleep and fatigue management), and ensure effective communication. Noise characteristics depend on amplitude, frequency spectrum, and duration. Typically, audio frequencies range from 15 Hz to 20 kHz; frequencies for voice communication occupy the 200 Hz–6 kHz band, while the ear is most sensitive in the 500 Hz–4 kHz range. Most attention is directed toward wide-band continuous noise between 22.4 Hz-11.2 kHz and disruptive short-term impulses.
Because there are no significant changes in human auditory physiology in microgravity, existing terrestrial standards are considered adequate for spacecraft acoustic design.
On the ISS, however, noise forms part of a continuous 24-hour operational environment. Therefore, standard workplace limits are not directly applicable and stricter criteria are implemented.

To compare data from different sources, sound pressure is expressed in decibels (dB) as:

$$P_{sdB} = 20 \log_{10} \left(\frac{P_s}{20} \right)$$

According to NASA-SPP-41000, hearing protection is required at continuous noise levels of 85 dB or greater. Infrasound in the 1–16 Hz range should not exceed 120 dB over a 24-hour exposure period [34].

Table 5.1: Limits of High-Frequency and Ultrasonic Noise [34].

Center Frequency [kHz]	**Noise Level [dB]**
10, 12.5, or 16	80
20	105
25	110
31.5 or 40	115

Vibrations

Vibration is generally divided into two categories: whole-body vibration and hand-arm vibration, depending on the part of the body exposed. Whole-body vibration, especially when experienced over extended periods, produces the most severe physiological and performance effects. It can negatively influence human efficiency, coordination, and task accuracy, with performance degrading as vibration frequency, amplitude, and duration increase.

In spacecraft environments, vibration sources include rotating machinery, pumps, and vehicle propulsion systems. These oscillations can combine with noise to intensify adverse effects such as stress, fatigue, and reduced cognitive and motor performance. Long-term exposure may also contribute to musculoskeletal strain and discomfort, particularly when combined with microgravity-induced physiological changes.

Mitigation of vibration effects requires careful engineering solutions. Among the most effective approaches are adjusting equipment operating parameters, selecting materials with improved damping properties, introducing flexible mounts or isolators, applying proper lubrication, and incorporating active or passive damping systems in the spacecraft's structure [34].

Metabolism

From a thermodynamic perspective, humans are open systems that continuously exchange matter and energy with their environment. The primary inputs include food, water, and oxygen, while the outputs consist mainly of carbon dioxide (CO_2), sweat, urine, and faeces. To sustain life, these basic requirements must be continuously satisfied. Figure 5.4 illustrates typical values of human metabolic inputs and outputs, based on an average metabolic rate of 137 W per person and an energy intake of approximately 2700 kcal per day. These values naturally increase with higher levels of physical activity or exercise.
The nominal oxygen consumption per person is approximately 0.84 kg per day, although it can range from 0.5 to 1.4 kg depending on individual metabolism and workload. Similarly, the average CO_2 production rate is about 1 kg per person per day, varying between 0.65 and 1.5 kg. In addition to metabolic demands, potable water is required for drinking, hygiene, and washing. The total daily water requirement per person is substantial (about 26.99 kg, representing roughly 90% of the total consumable mass). For long-duration missions, the design and operation of an efficient water recycling system are therefore essential, as the total mass of consumables otherwise becomes prohibitively large. Humans also produce heat as a by-product of metabolism, and this heat generation rate varies according to both individual characteristics and activity level [34].

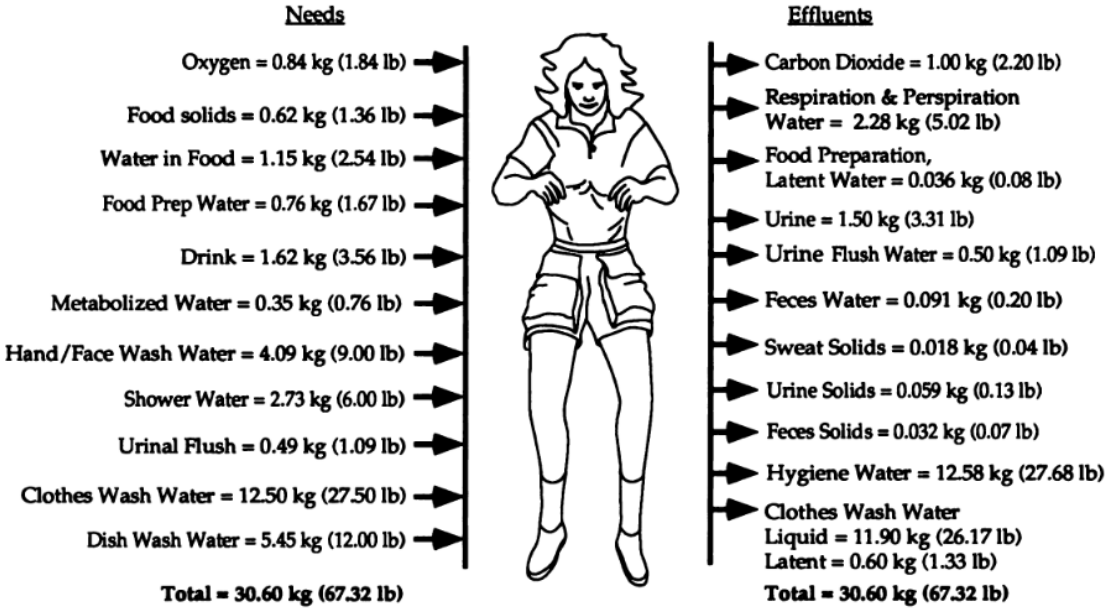


Figure 5.4: Human metabolic input and output per person per day [37].

Microgravity

Under Earth's gravity, body fluids are pulled downwards, accumulating in the lower extremities where higher blood pressure counterbalances the lower pressure in the upper body. When gravity is removed, this hydrostatic gradient is nullified, and fluids redistribute uniformly throughout the body, a process known as fluid shift. This leads to facial oedema, nasal congestion, and a reduction in leg volume, a condition often referred to as the "chicken leg" syndrome. As fluid volume in the lower body decreases, the heart encounters less vascular resistance when pumping blood, resulting in a variations of heart rate, central venous pressure, and stroke volume. Over time, these cardiovascular adaptations can lead to gradual cardiac atrophy and a reduction in overall aerobic capacity.
Microgravity also affects the vestibular and proprioceptive systems. The absence of a constant gravitational reference disturbs the normal coordination between visual, vestibular, and somatosensory inputs, frequently causing space motion sickness (SMS). Symptoms typically include nausea, dizziness, and disorientation during the initial days of adaptation to weightlessness.
Prolonged exposure to microgravity further reduces the mechanical loading on bones and muscles. Skeletal tissues experience accelerated bone demineralisation and calcium excretion, leading to a form of disuse osteoporosis.
Similarly, muscle fibres, particularly those involved in postural support, labelled in Figure 5.5, undergo atrophy due to the lack of gravitational resistance, as show in Table 5.2. [38, 39].

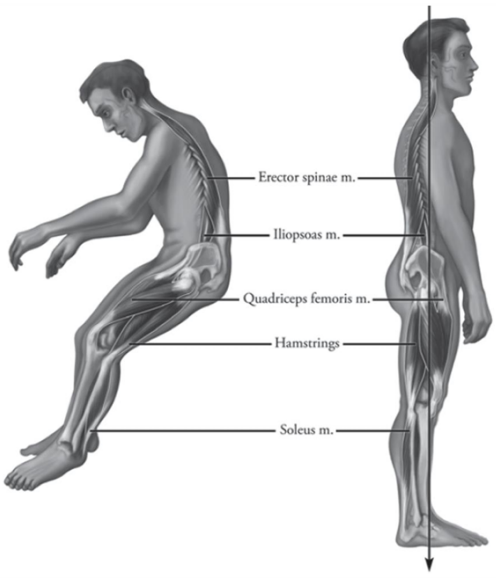


Figure 5.5: Postural shift in microgravity [38].

Table 5.2: Volume changes on landing day (after 6-months flight) [38].

Muscle group	**% Loss after long-duration flight**
Back (erector spinae and intrinsic)	-10.9
Iliopsoas	-20.0
Quadriceps	-12.1
Hamstrings	-15.7
Soleus	-19.6

For short-term missions, regular in-flight exercise has proven effective in mitigating most physiological changes. For long-duration missions, additional countermeasures have been proposed, such as artificial gravity systems (0.5 - 0.8 g). According to NASA-STD-3000, crew members on missions lasting more than 10 days should engage in structured physical exercise to counteract these adverse effects [34].

Psychological effects

Space travel, particularly long-duration missions, exposes astronauts to multiple stressors and psychological challenges. Beyond weightlessness, they must manage interpersonal relationships, social isolation, and confinement in noisy, restricted environments. Common stress-related symptoms include hostility, conflict, anxiety, asthenia, insomnia, and depression. These conditions impair attention, memory, mood, and performance, and are difficult to diagnose because symptoms overlap. Even a single affected crew member can jeopardise mission success, as demonstrated by the early termination of the Salyut 7 mission.
Analogue environments, such as Antarctic stations, undersea habitats, and space simulators, provide insights into human behaviour under these conditions. Real-time space-to-ground communication is crucial for psychological support, though future Mars missions will face delayed communication and the "Earth out-of-view" phenomenon.
Interpersonal conflict is a major psychological issue. Person-to-person tensions arise from unresolved arguments, cross-cultural differences, gender dynamics, and leader–follower relationships, sometimes posing a greater risk to mission success than physiological factors. Conflicts with ground control, or the "Us vs. Them" syndrome, occur when crews perceive command as out of touch, often over scheduling and flexibility issues. Identifying a common adversary, such as the control centre, can paradoxically enhance crew cohesion and collaboration.

Table 5.3: Reported problems on space missions and analogues [38].

Reported Problems	**ISS/Mir**	**Shuttle**	**Submarine**	**Polar Exped.**
Interpersonal conflicts	D	D	D	D
Sleep disturbances	D	D	D	D
Boredom, restless	A	-	D	D
Lower Performance	A	-	D	D
Lower Compatibility	A	A	A	D
Substance abuse	A	-	-	D

D: documented; A: anecdotal

5.3 Environmental Control and Life Support System (ECLSS)

As stated in the European Cooperation for Space Standardization (ECSS), the ECLS is an:

"engineering discipline dealing with the physical, chemical and biological functions to provide humans and other life forms with suitable environmental conditions. The objective of ECLS is to create a suitable environment by controlling the environmental parameters, providing resources, and managing waste products. It must also support special operations such as EVA, respond to environmental contingencies and provide health related services" [40].

In human spaceflight, the Environmental Control and Life Support System (ECLSS) is an essential part of any mission, ensuring that spacecraft, stations, or habitats maintain physiologically acceptable environmental conditions. As noted in Section 5.1, the human organism is an *open system* that continuously exchanges energy and matter with its surroundings.
The ECLSS must therefore supply the crew with the necessary resources and manage their by-products while maintaining an environment suitable for human life. It must also support activities such as hygiene, medical procedures, and scientific operations. To meet these needs, the ECLSS manages the atmosphere, water, waste, and food [35].

Typically, the ECLSS is divided into several functional areas [35, 41]:

- Atmosphere management: atmospheric composition control, temperature and humidity control, pressure control, atmosphere regeneration, contamination control and ventilation
- Water management: provisioning of potable and hygiene water, recovery and processing of waste water
- Food production and storage: production and storage of food
- Waste management: collection, storage, and processing of human waste and other refuse
- Crew safety: fire detection and suppression, and radiation shielding

These functions and related subsystems are well illustrated in Figure 5.6. They can generally be classified into non-regenerative functions, which do not involve recycling, and regenerative functions, which recover and reuse life-support resources [41].

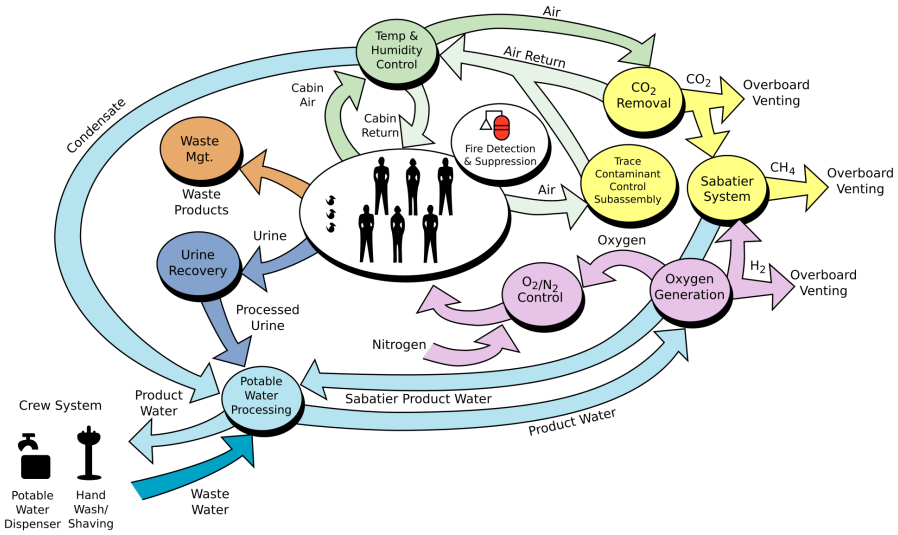


Figure 5.6: ECLSS flow diagram [42].

The configuration of an environmental control and life support system is heavily influenced by technological maturity, human factors, mission profiles, safety requirements, and cost constraints. Designing an ECLSS architecture is thus an iterative process that must compare different technologies and configurations against mission-specific needs.
An ECLSS that relies entirely on stored resources, without any recycling capability, is referred to as an *open loop* system, whereas an ECLSS incorporating resource recovery is described as a *closed loop* system [41].

In open loop systems, matter continuously flows into and out of the spacecraft: food, water, and oxygen are entirely supplied from storage, and the quantities resupplied must match the crew's consumption. These systems are simple, robust, and have been widely used in human spaceflight, but their main limitation is that resource mass increases linearly with mission duration and crew size [41].

Closed loop systems instead bring an initial supply from Earth and process waste to recover useful resources. As loop closure increases, resupply needs decrease. Their advantage is that most mass is transported only once, initial consumables and processing hardware, while only small make-up quantities are needed to replace unrecoverable losses. The drawbacks include lower technological maturity and higher power and thermal loads [41].

For instance, Apollo spacecraft operated with open loop systems, whereas the International Space Station relies on a predominantly closed loop architecture [35].

Table 5.4: Reduction of relative supply mass by successive loop closure [41].

Step	**Method**	**RSM** [%]
0	Open loop	100
1	Waste water recycling	45
2	Regenerative carbon dioxide-absorption	30
3	Oxygen recycling from carbon dioxide	20
4	Food production from recycled wastes	10
5	Elimination of leakage	5

RSM: Relative Supply Mass

As regenerative technologies are progressively implemented in life support systems, the requirement for resupply mass decreases substantially. As shown in Table 5.4, incorporating waste water recycling alone reduces the relative supply mass by over 50% compared with an open-loop system. However, further increases in closure level continue to reduce resupply mass while simultaneously raising system complexity, costs, and energy consumption, and diminishing overall reliability [35].

5.4 The ECLSS on the ISS

The ISS ECLSS consists of a network of interconnected subsystems, each responsible to ensure crew survival and habitable conditions. These functions include atmospheric control and supply, temperature and humidity regulation, wastewater management and potable water provision, fire detection and suppression, and atmosphere revitalisation.
Following NASA standards, applicable to the ISS in particular, the ECLSS can be subdivided into a series of subsystems. These are later listed in Table 5.5.

Atmosphere Control and Supply (ACS)

This subsystem supplies oxygen and nitrogen to maintain the Space Station atmosphere at the appropriate pressure and composition for human habitation. It also provides gas support to various ISS users, as well as pressure equalisation and depressurisation capabilities.
Oxygen is primarily generated by the Elektron system, which electrolyses water into hydrogen and oxygen. Additional oxygen can be supplied by Solid Fuel Oxygen Generators, which use chemical cartridges in an exothermic reaction. Oxygen and nitrogen for the USOS ACS are managed through dedicated supply, distribution, and control elements.
Four high-pressure gas tanks, two for oxygen and two for nitrogen, are mounted externally on the Airlock. These gases are distributed throughout the USOS via a plumbed network, while a separate high-pressure system enables tank recharge by the Shuttle.
The Pressure Control Assembly (PCA) monitors cabin pressure, regulates the injection of oxygen and nitrogen, and enables depressurisation of Station volumes. Depressurisation is used both during nominal operations to relieve overpressure and in emergency situations to vent hazardous contaminants or, as a last resort, to suppress fires [43].

Temperature and Humidity Control (THC)

This subsystem contributes to maintaining a habitable environment within the ISS by circulating cool, dry air, controlling temperature and humidity, and removing airborne particulates. Atmospheric circulation reduces temperature gradients, ensures a homogeneous gas composition, and supports smoke detection. Ventilation is provided at three levels: rack, intramodule, and intermodule.
Rack ventilation cools and circulates air within individual equipment racks. Intramodule ventilation ensures uniform atmospheric conditions within a single module and supports temperature regulation and humidity removal. Intermodule ventilation circulates air between modules, maintaining a consistent atmosphere

throughout the Station.
The Avionics Air Assembly (AAA) provides cooling and air circulation within specific rack volumes. It consists of a fan and a non-condensing heat exchanger, which cools avionics equipment and supports the operation of smoke detectors in the Fire Detection and Suppression Subsystem. The heat exchanger interfaces with the Internal Thermal Control System (ITCS).
The Common Cabin Air Assembly (CCAA) provides intramodule ventilation, temperature control, and humidity removal through a fan and a condensing heat exchanger. Prior to entering the CCAA, air passes through High Efficiency Particulate Air (HEPA) filters that remove particulates and bacteria. Condensed moisture is collected and routed to the Water Recovery and Management (WRM) subsystem, while the cooled and dehumidified air is supplied to the Atmosphere Revitalization Subsystem, specifically to the Carbon Dioxide Removal Assembly (CDRA), to support efficient CO_2 removal.
Intermodule ventilation is provided by the Intermodule Ventilation (IMV) Assembly, which consists of fans and valves that drive airflow between modules via a ducting network [43].

Atmosphere Revitalisation (AR)

This subsystem ensures that the cabin atmosphere remains safe and comfortable for breathing.
The Major Constituent Analyser (MCA) monitors the composition of the Station atmosphere using mass spectrometry. Its measurements are used to regulate the addition of oxygen and nitrogen and to assess the performance of the carbon dioxide removal system.
The Carbon Dioxide Removal Assembly (CDRA) removes carbon dioxide from the cabin atmosphere using a series of regenerable sorbent beds and vents the extracted gases to space. Effective CO_2 removal requires cold, dry air; therefore, the CDRA receives conditioned air from the Temperature and Humidity Control (THC) subsystem and interfaces directly with the Internal Thermal Control System (ITCS).
The Trace Contaminant Control Sub-assembly (TCCS) removes trace gaseous contaminants and odours from the cabin atmosphere through filtration and catalytic oxidation. These contaminants originate from material off-gassing, leaks, spills, and other onboard activities [43].

Waste Management (WM)

The Waste Management Subsystem comprises a commode that operates by drawing cabin air to direct faeces into a waste canister, where they are collected in bags and subsequently compacted by a piston.
Urine collection is achieved through a funnel assisted by directed cabin airflow. The urine is pre-treated with sulphuric acid (H_2SO_4) before being routed to the urine processor [37].

Water Recovery and Management (WRM)

This subsystem collects, stores, processes, and distributes the Station's water resources. Water sources include condensate recovered by the Temperature and Humidity Control (THC) subsystem and wastewater returned from Extravehicular Activity (EVA) operations.
Collected water is purified and continuously monitored for quality. When used by the Elektron oxygen generator, potable water may require additional purification to remove dissolved minerals. Water is stored and transported throughout the USOS using small tanks and dedicated distribution lines.
Solid waste generated by various onboard activities is collected and transferred to a Progress module for disposal by incineration during atmospheric re-entry.
Wastewater lines route condensate and EVA wastewater to storage tanks, where it is held until further processing or removal from the system [43].

Fire Detection and Suppression (FDS)

The Fire Detection and Suppression Subsystem provides smoke detection for Station volumes, portable fire extinguishers, portable breathing equipment, and alarm systems with automatic software responses to fire events. After a fire, the Atmosphere Revitalisation (AR) and Temperature and Humidity Control (THC) subsystems operate together to remove contaminants from the affected volume. In extreme cases, the Atmosphere Control and Supply (ACS) subsystem can depressurise a module to extinguish the fire and/or vent hazardous contaminants.
Upon smoke detection, flight software automatically shuts down local THC equipment to limit oxygen supply to the fire. Crew members can manually activate or silence fire alarms using the Caution and Warning (C&W) Panel or the Portable Computer System (PCS).
Fires within the USOS are extinguished using handheld Portable Fire Extinguishers filled with carbon dioxide [43].

Food Storage

Food storage within the ISS is based on a combination of pre-packaged, long-shelf-life products and a limited supply of fresh food to enhance crew diet variety. Most stored food consists of ambient-stable items, including freeze-dried, thermostabilised, low-moisture, and irradiated products, packaged in individual portions for ease of use in microgravity. While the majority of food is stored in stable forms, fresh items such as fruits and vegetables are periodically supplied to mitigate menu fatigue. Food storage is closely integrated with preparation and consumption systems, ensuring compatibility with onboard facilities and efficient use of limited space and resources [43].

Crew Health Care (CHeC)

The Crew Health Care Subsystem is dedicated to maintaining the overall health of astronauts and is composed of three main subsystems: the Countermeasures System (CMS) provides the equipment and protocols required for daily exercise, aimed at mitigating the physiological effects of prolonged exposure to microgravity; the Environmental Health Subsystem (EHS) monitors atmospheric conditions for contaminants arising from crew and station activities, as well as water quality and radiation levels; the Health Maintenance System (HMS) supports in-flight preventive medicine, diagnostic and therapeutic care, and routine treatment for the majority of medical conditions expected during ISS operations [43, 44].

EVA Support

The Air Lock (AL) is a critical element for supporting extravehicular activities (EVAs), providing capabilities for depressurisation, egress, ingress, and repressurisation [43].
EVAs are generally classified into three categories: scheduled, unscheduled, and contingency.
Scheduled EVAs are planned as part of the nominal mission timeline, whereas unscheduled EVAs are conducted to achieve or enhance mission objectives but are not initially included in the flight plan.
Contingency EVAs, on the other hand, are performed in emergency situations to ensure crew and vehicle safety.
In the context of the ISS, EVAs are typically categorised into only two types: nominal and contingency. This distinction arises from the operational flexibility of the Station, where unscheduled EVAs can be incorporated into the nominal mission plan [44].

Table 5.5: ECLSS subsystems [35, 44, 45, 46].

Subsystem	**Implemented function**
Atmosphere Control and Supply (ACS)	• Atmosphere Constituents Storage • Total Atmospheric Pressure Control • Oxygen Partial Pressure Control • Overpressure Relieve • Pressure Equalisation • Rapid Decompression Response • Hazardous Atmosphere Response
Temperature and Humidity Control (THC)	• Atmospheric Temperature Control • Atmospheric Moisture Control • Ventilation • Microorganisms and Airborne Particulate Contaminants Control • Equipment Cooling • Thermally Conditioned Storage
Atmosphere Revitalisation (AR)	• CO_2 Removal and Reduction • O_2 Generation • Gaseous Contaminants Control • Major Constituents Monitoring
Waste Management (WM)	• Metabolic Waste Management • Other Solid Waste Management • Liquid/Gaseous Waste Management

Subsystem	**Implemented function**
Water Recovery and Management (WRM)	• Hygiene Water Supply • Potable Water Supply • Water Storage • Urine Processing • Waste Water Processing • Water Quality Monitoring
Fire Detection and Suppression (FDS)	• Fire Detection • Fire Isolation • Fire Suppression
Food Storage	• Food Preservation • Food Stowage • Nutritional Tracking
Crew Health Care (CHeC)	• Health Monitoring • Medical Diagnostic • Emergency Medical Care
EVA Support	• Denitrogenation Support • Service and Checkout Support • Station Egress and Ingress Support

In parallel with the definition of system functions, Table 5.6 identify the hardware items that implement and fulfil the mentioned functions across the various subsystems.

Table 5.6: Major USOS ECLS hardware items [43, 46].

Subsystem	**Capability or Device**
Atmosphere Control and Supply (ACS)	• Pressure Control Assembly (PCA) • O_2/N_2 Storage Tanks • Pressure Equalisation Valves
Temperature and Humidity Control (THC)	• Common Cabin Air Assembly (CCAA) • Avionics Air Assembly (AAA) • Intermodule Ventilation (IMV)
Atmosphere Revitalisation (AR)	• CO_2 Removal Assembly • CO_2 Reduction Assembly • Major Constituents Analyser (MCA) • Oxygen Generator Assembly (OGA)
Waste Management (WM)	• Commode/Urinal
Water Recovery and Management (WRM)	• Water Processor (WP) • Water Storage • Urine Processor (UP)
Fire Detection and Suppression (FDS)	• Smoke detectors • Fire suppressors

Subsystem	**Capability or Device**
Food Storage	• Refrigerators • Ovens • Food trays
Crew Health Care (CHeC)	• Advanced Life Support Pack • Ambulatory Medical Pack • Crew Medical Restraint System • Respiratory Support Pack
EVA Support	• Extravehicular Mobility Unit • Orlan-M

5.4.1 Physico-chemical vs Bioregenerative Technologies

The design of spacecraft life support systems is driven by the need to maintain, within an isolated volume, an environment that is suitable for the health and well-being of both the crew and onboard systems for the entire duration of the mission. The selection of appropriate technologies for a given application therefore depends largely on mission characteristics such as crew size, mission duration, mission location, and the availability of resources [41].

For this reason, the choice of suitable life support technologies is a critical aspect of system design. Life support technologies are generally classified into two main categories: *physico-chemical* (P/C) and *bioregenerative*. Physico-chemical systems rely on mechanical and chemical processes such as fans, filters, physical or chemical separation, and concentration processes. Bioregenerative systems, on the other hand, make use of living organisms, including plants and microorganisms, to regenerate essential resources. Systems that combine both approaches are referred to as *hybrid life support systems* (HLSS) [47].

Table 5.7: Comparing Physico-chemical and Bioregenerative technologies [47].

Functions	**P/C solution**	Bio solution
Pressure control	Valves, regulators, and heaters with control algorithms.	Unprovided.
Temperature and Humidity control	Internal heat loads and external heat fluxes transferred to a water coolant loop.	Unprovided.
Ventilation	Fan, ducting, and isolation valves with different power consumption, noise, and vibrations.	Unprovided.

Functions	**P/C solution**	**Bio solution**
CO_2 removal	4-BMS[1], 2-BMS, and electrochemical depolarisation concentration to remove CO_2 from the loop.	Plants, algae, and bacteria to fix carbon.
Monitoring atmosphere	Sensors, mass spectrography, and gas chromatography to monitor the atmosphere composition.	Unprovided.
Makeup oxygen	Electrolysis or CO_2 reduction to produce oxygen.	Plants and algae to fix carbon and to produce oxygen.
Water provisioning	Tanks to store water, chemical processor to produce water from wastewater and urine.	Bioreactor to produce water form bacteria action, plants root to filter wastewater.
Food provisioning	Storage to prolong food duration.	High plants to produce food in-situ.

[1]4-bed molecular sieve: 2 adsorbing beds, 1 desiccant bed, one zeolite molecular sieve

5.5 Modelling the ECLSS

In this modelling activity, the primary focus is placed on the ECLS subsystem, which, by definition, operates alongside several other subsystems within the ISS architecture. In the present model, as seen in Figure 5.7, these additional subsystems are included only as placeholders, serving to represent virtual interfaces and connections between blocks.

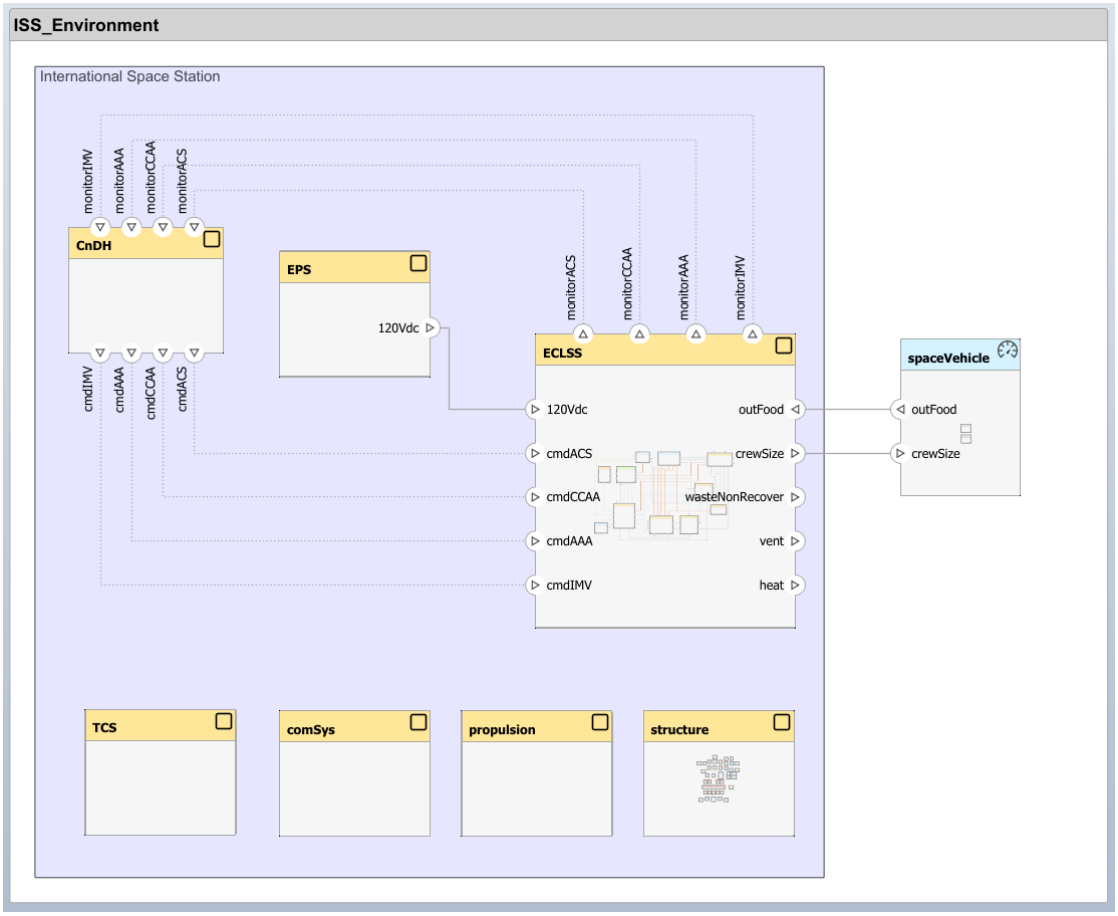


Figure 5.7: ISS Environment scheme.

A *yellow* colour is used to identify the various subsystems through stereotypes, while *light blue* is used to represent a hypothetical external spacecraft docking with the ISS.

Particular attention can be given to the representation of the ISS modularisation within the *Structure* subsystem, which effectively emphasises the purpose of MBSE. In this example, the approach allows the entire station structure to be represented within a single architectural block.

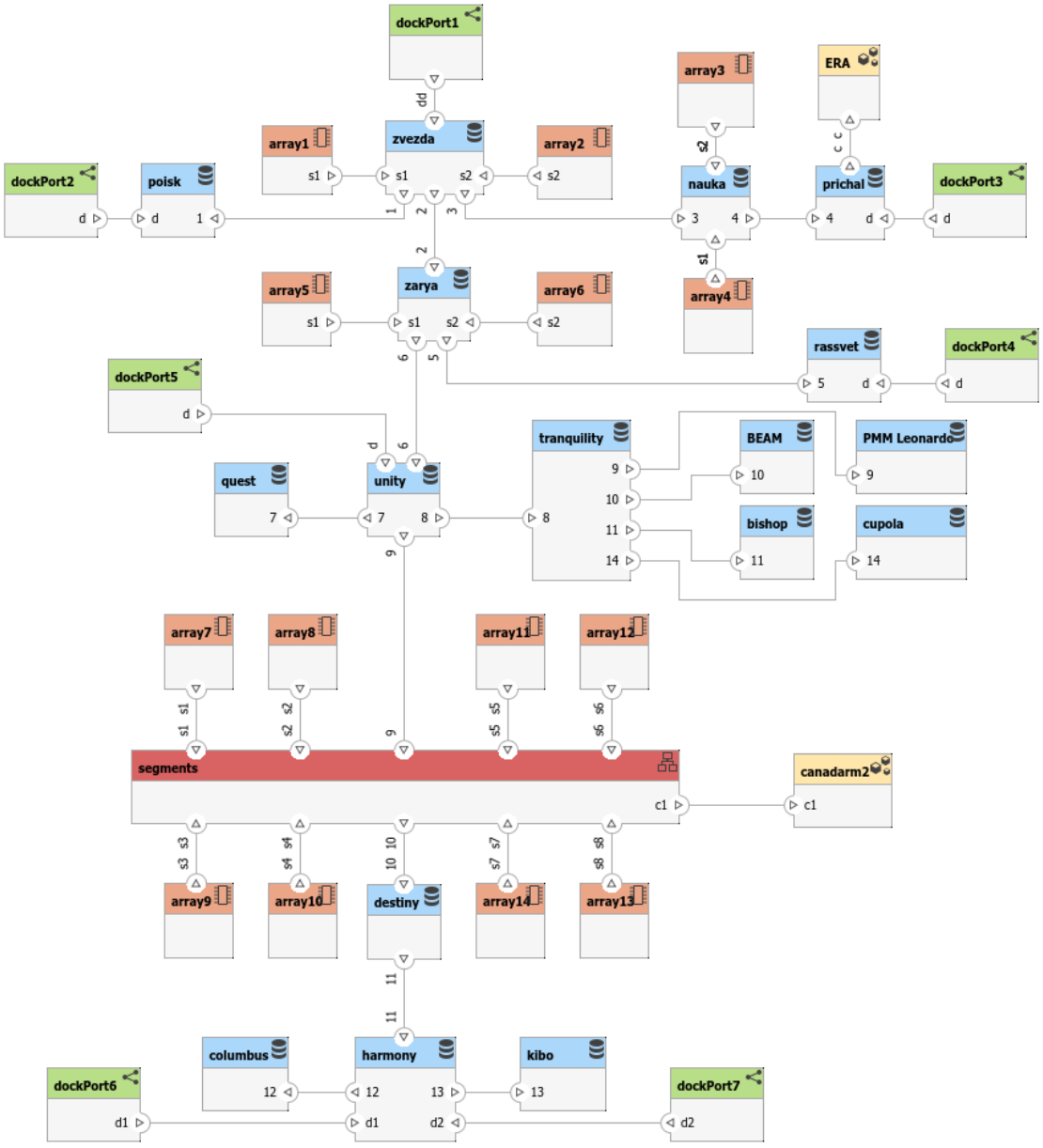


Figure 5.8: ISS block view.

5.6 Workflow and Characterisation

For the correct development of the subsystem architecture, it was essential to follow established guidelines and reference schemes. The two frameworks adopted in this work are shown in Figure 5.9 and Figure 5.10.

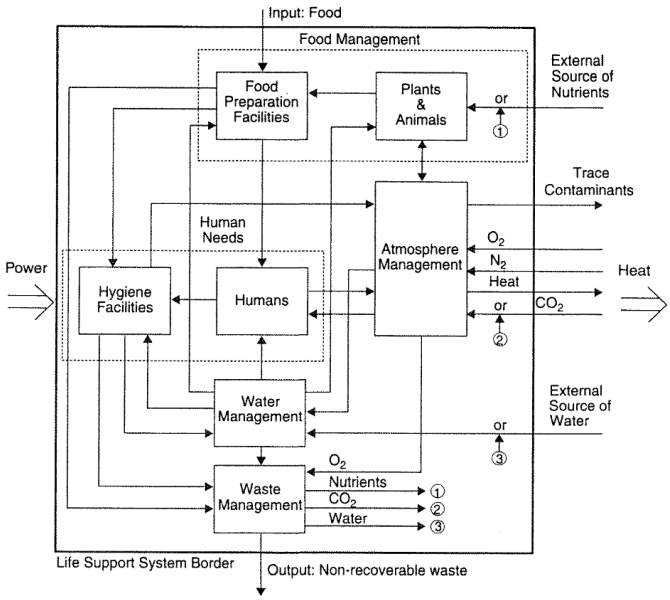


Figure 5.9: Life Support Functions and Relationships [47].

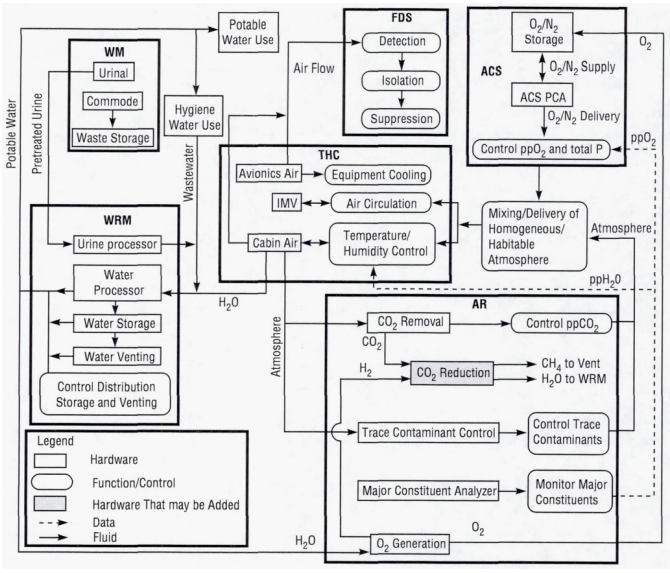


Figure 5.10: USOS ECLS functional integration [37].

Figure 5.12 presents the result of the analysis of the two macro-schemes implemented in System Composer. The diagram shows a set of blocks characterised by stereotypes, which are used to highlight and distinguish the intrinsic complexity of the system. The *green* blocks represent the sole payload of the ECLSS, namely the crew. The subsystems are again identified in *yellow*, consistently with the higher levels of the architecture. Lastly, the *light blue* blocks denote other facilities, support systems, or entities that interface with the architecture without constituting full subsystems, such as EVA operations or the cabin atmosphere.

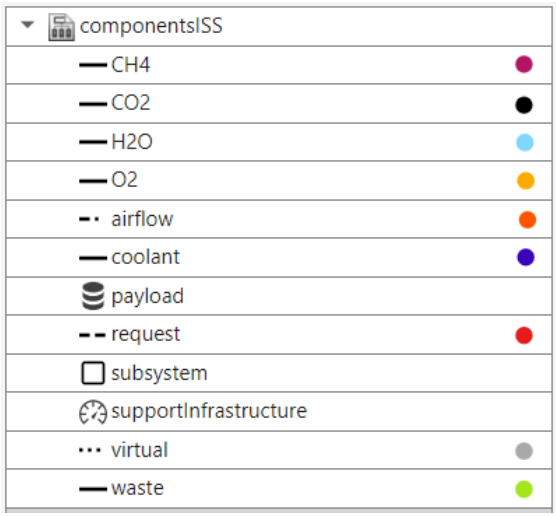


Figure 5.11: ECLSS Applied Stereotypes.

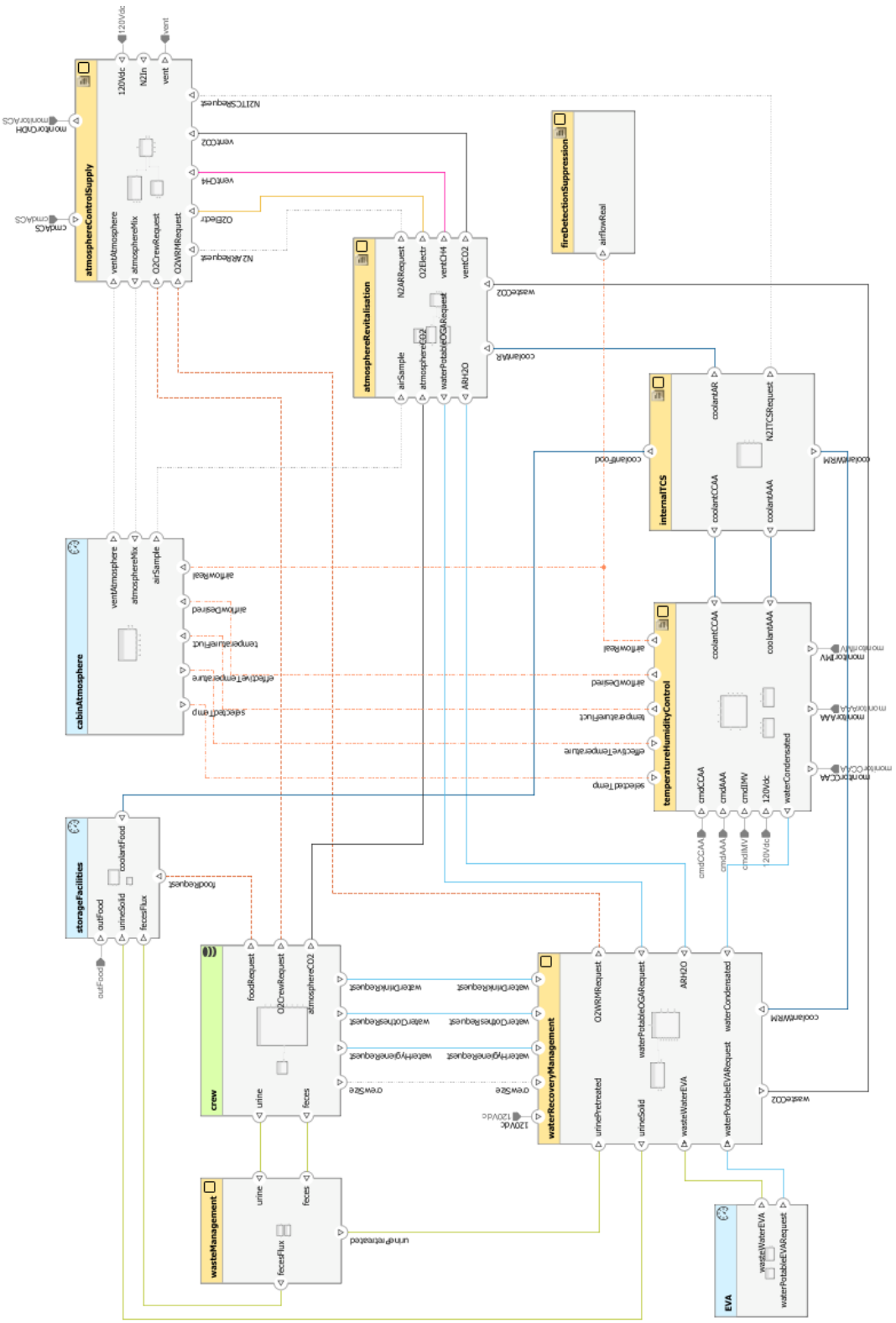


Figure 5.12: Final block architecture for ECLSS within System Composer.

5.6.1 Simulation Timing

In order to fully exploit the modularity offered by System Composer in combination with the temporal execution framework of Simulink, a modelling convention was adopted where one second of simulation time corresponds to one hour of real time. This approach allows a full day of operations aboard the space habitat to be represented within a 24-second simulation window. As a consequence, blocks operating on a continuous-time basis were excluded in favour of discrete-time blocks, which are more suitable for this abstraction level.
In particular, the use of a *Pulse Generator* block (amplitude: 1, period: 24, pulse width: 0.01, phase delay: 24) enabled the generation of time-scheduled signals consistent with operational needs. These signals are used to trigger the demand and production profiles associated with the human payload, ensuring a coherent temporal representation of crew-related processes within the simulation.

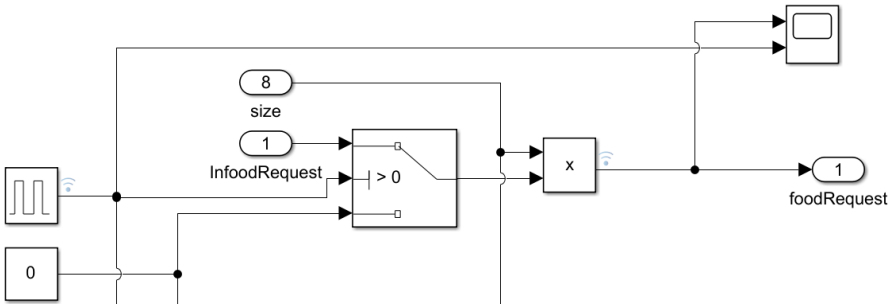


Figure 5.13: Simulink Architecture for Daily Food Request Pulse.

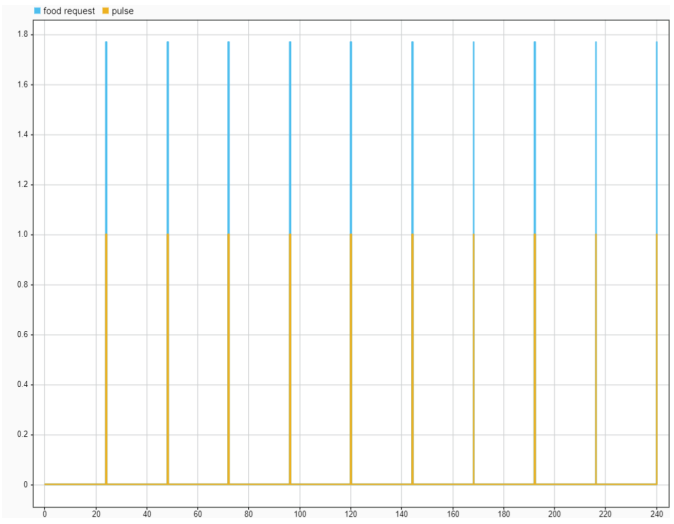


Figure 5.14: Timing and Food Request Pulse diagram (1 crew member, 10 days).

5.6.2 ECLSS Payload: Crew

With respect to the crew, the first parameter to be defined within the model is the number of occupants of the space habitat. As previously discussed in Subsection 5.2, averaged daily physiological requirements per individual were adopted as reference values. Varying the crew size directly affects the demand for consumables and the production of metabolic by-products, and represents a key driver for system sizing and trade-off analyses.
To support this parametrisation, a dedicated *Constant* block was introduced in the System Composer model. As shown in Figure 5.13, through signal connections, this block scales the daily consumption and production rates associated with a single crew member, enabling a modular and scalable representation of the human payload within the system architecture.
This modelling approach allows the crew-related parameters to be propagated across multiple subsystems, where relevant variables are generated and updated dynamically.

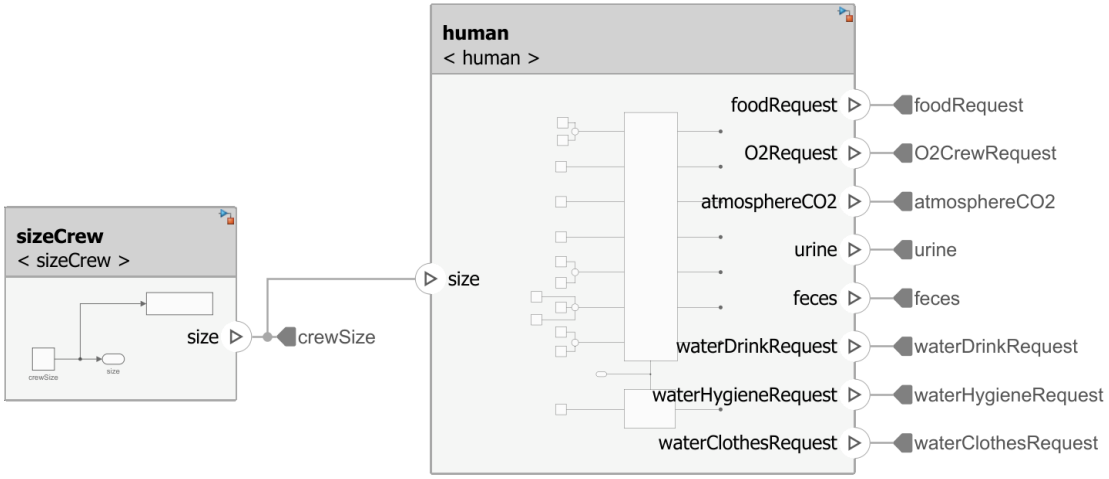


Figure 5.15: System Composer Architecture for ECLSS *Crew*.

5.6.3 ECLSS Subsystems

The following examination provides a more detailed description of the various functions performed by ECLSS. It is essential to recognise that these functions depend strongly on the mission scenario, therefore different mission durations, destinations, and crew profiles impose markedly different environmental and metabolic requirements [35, 37].

ACS Subsystem

The composition of the atmosphere within a space habitat is determined during the design phase on the basis of several physiological requirements (see Section 5.2). The ACS is required to regulate both the composition and the pressure of atmospheric gases to ensure optimal crew performance and the proper functioning of onboard equipment. In a space habitat, oxygen is continually consumed by human metabolism and other operational processes, while portions of the cabin atmosphere are lost through leakage, airlock cycling, or experimental activities. Consequently, the resupply of O_2 and N_2 is necessary to restore the cabin atmosphere to its nominal conditions.
The ACS is also responsible for preventing overpressure and for responding appropriately to rapid decompression events [46].

For the modelling of this subsystem, a simplified configuration was adopted by considering the presence of only two storage tanks, one for O_2 and one for N_2. In particular, the oxygen tank was modelled in greater detail, based on the specifications reported by Wieland [37]. Specifically, the oxygen tank receives as inputs the signals *O2CrewRequest* and *O2WRMRequest*, which represent the oxygen demand arising from human metabolic requirements and from the operation of the Water Processor (WP) within the Water Recovery and Management (WRM) system, respectively, as well as *O2Electr*, corresponding to the internal oxygen production generated by the electrolyser located in the Atmosphere Revitalisation (AR) system. In addition, a stochastic variability term was introduced to represent potential intrinsic leakage phenomena within the tank.

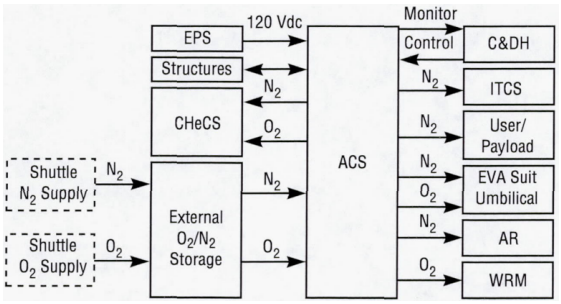


Figure 5.16: Wieland ACS specifications [37].

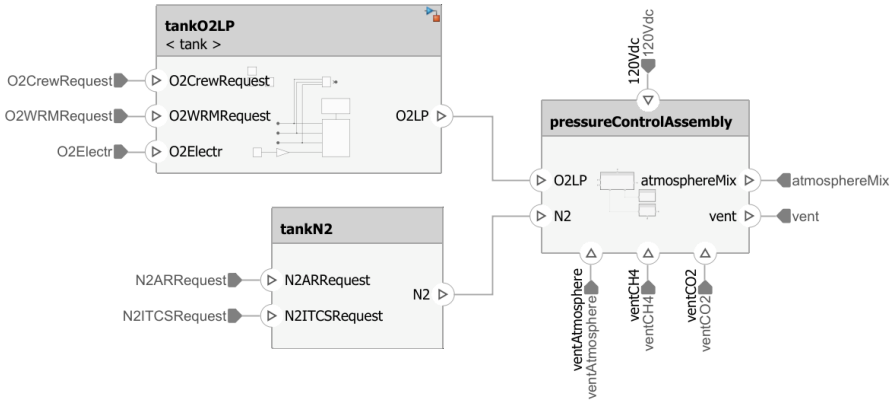


Figure 5.17: ACS within System Composer.

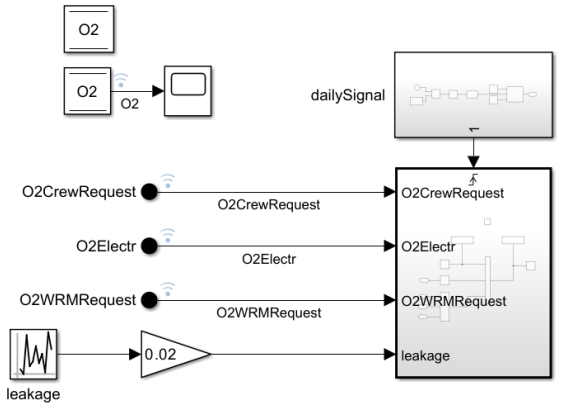


Figure 5.18: TankO2LP zoom in.

The temporal evolution of the signals can be analysed in Figure 5.19 and Figure 5.20.

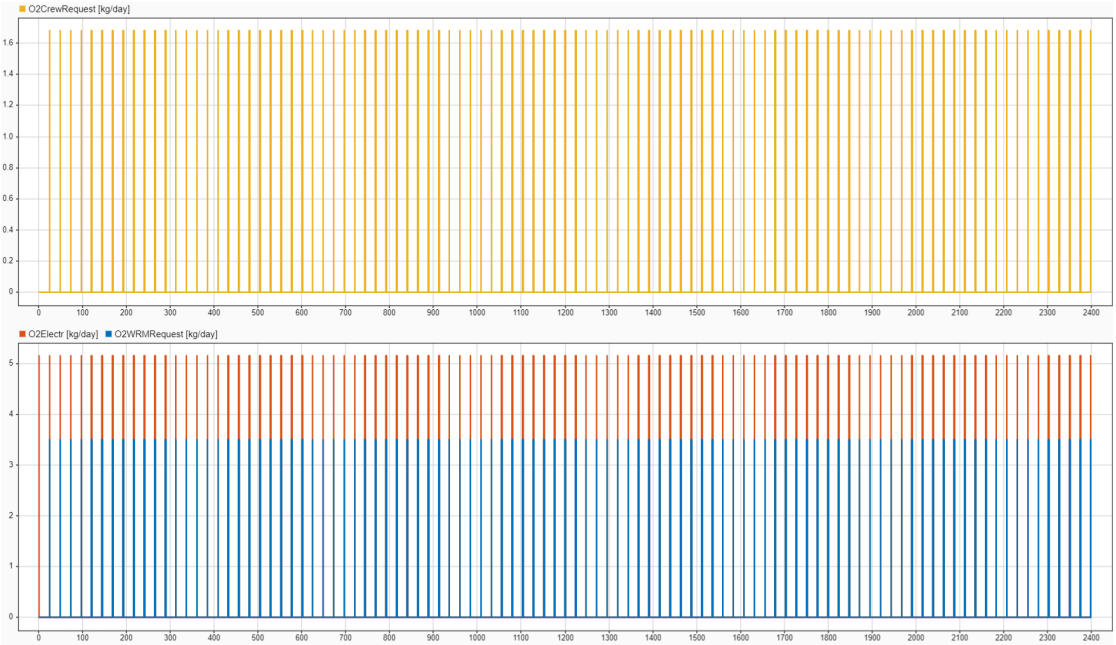


Figure 5.19: O2CrewRequest, O2Electr, and O2WRMRequest signals temporal evolution (2 crew members, 100 days).

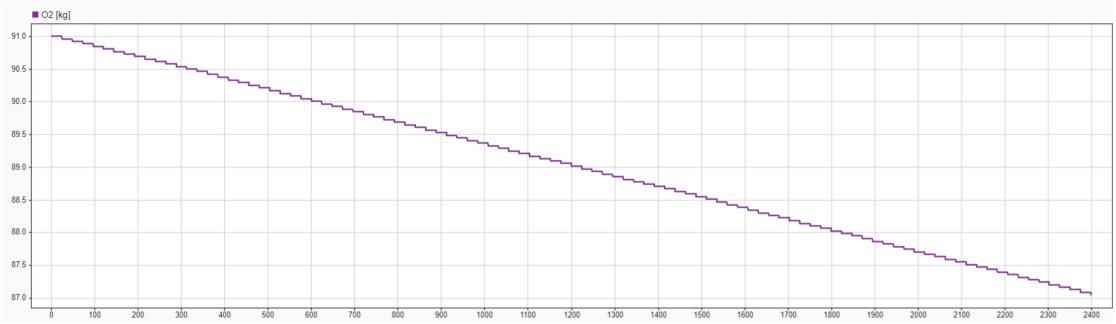


Figure 5.20: Oxygen tank depletion (2 crew members, 100 days).

The virtual connections originating from the two storage tanks lead to the definition of the Pressure Control Assembly (PCA). This component was modelled at a lower level of detail, while preserving the relevant design parameters. As shown in the internal decomposition of the Pressure Control Panel (PCP) within the PCA in Figure 5.21, the subsystem architecture explicitly represents its functional and structural elements.

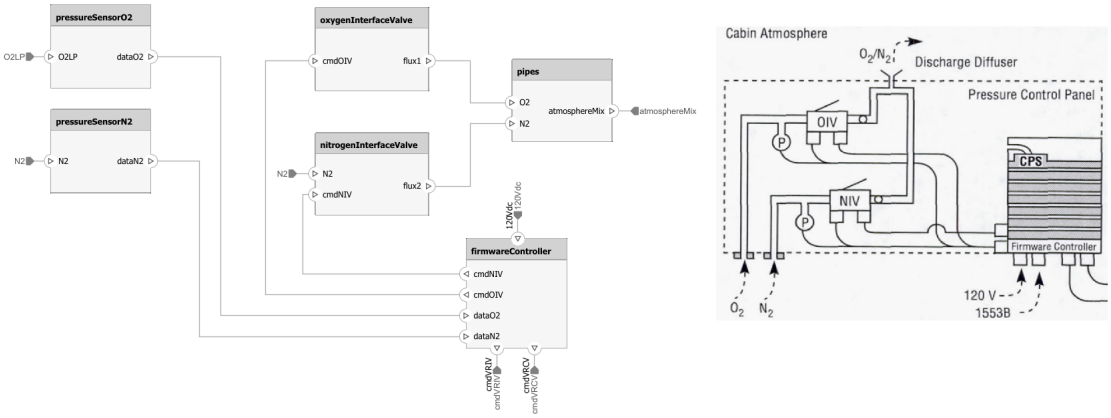


Figure 5.21: Pressure Control Panel zoom in.

As previously stated, System Composer supports the definition of requirements. Figure 5.22 presents a set of plausible requirements for the ACS subsystem.

1	R-SYS-ACS	Container	**ACS Requirements**	⇦Implemented by: atmosphereControlSupply
1.1	R-SYS-ACS-010	Container	**Control Pressures**	
1.1.1	R-SYS-ACS-011	Functional	**Control Total Pressure** The ACS shall control ISS atmospheric total pressure.	
1.1.2	R-SYS-ACS-012	Functional	**Control O2 partial pressure** The ACS shall control ISS atmospheric O2 partial pressure.	
1.1.3	R-SYS-ACS-013	Functional	**Control N2 partial pressure** The ACS shall control ISS atmospheric N2 partial pressure.	
1.2	R-SYS-ACS-020	Functional	**Internal Distribution**	
1.2.1	R-SYS-ACS-021	Functional	**O2 Distribution** The ACS shall distribute O2 to ISS system, crew, and payload interfaces.	
1.2.2	R-SYS-ACS-022	Functional	**N2 Distribution** The ACS shall distribute N2 to ISS system, crew, and payload interfaces.	

Figure 5.22: Example of ACS Requirements.

THC Subsystem

The THC subsystem is responsible for maintaining appropriate thermal and humidity conditions (see Section 5.2) by regulating atmospheric temperature, controlling humidity, providing ventilation, cooling on-board equipment and ensuring thermally conditioned storage.
The subsystem is also in charge of controlling microorganisms and airborne particulate contaminants [35, 37].

For this subsystem, it was possible to implement a complex interaction between the *temperatureHumidityControl* and *cabinAtmosphere* blocks, representing the THC and the cabin atmosphere of the habitat, respectively.
Focusing first on the THC block, Figure 5.23 shows the presence of three internal subsystems: the Common Cabin Air Assembly (CCAA), which constitutes the primary focus of this analysis, the Avionics Air Assembly (AAA), and the Inter-Module Ventilation (IMV).
The objective of this modelling effort is to simulate the demand for a specific *airflow* as a function of the effective cabin temperature, in accordance with the applicable design requirements reported in [37].

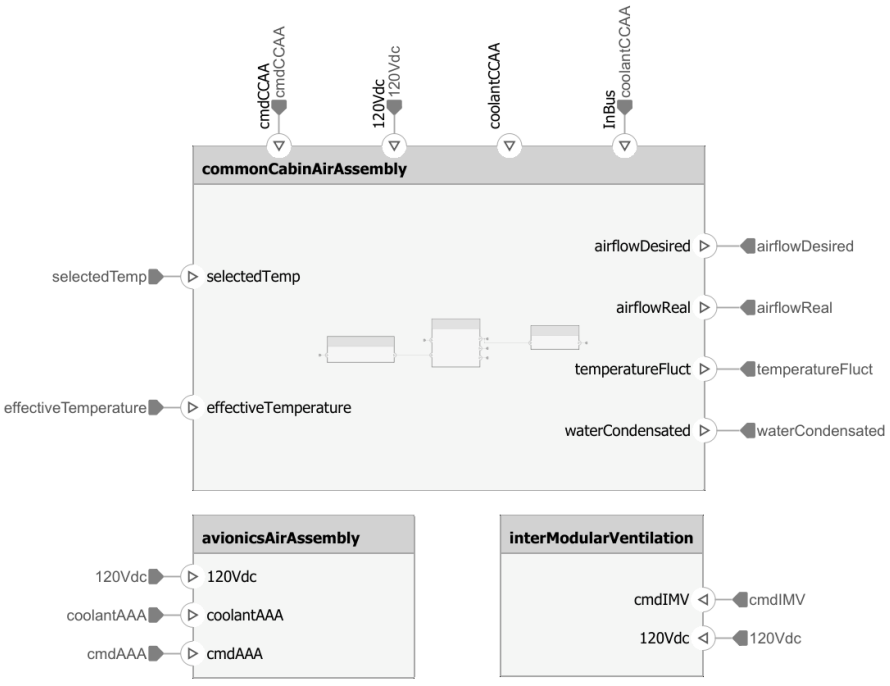


Figure 5.23: THC zoom in.

Within the CCAA, the following components were modelled: a Resistance Temperature Detector, which simulates a cabin temperature sensor, a Temperature Control Check Valve (TCCV), and a Condensing Heat Exchanger.
As shown in Figure 5.25, the TCCV incorporates an internal control logic that iteratively computes both the thermal equilibrium condition of the habitat and the corresponding target *airflow* toward which the system converges (Figure 5.28.
The Condensing Heat Exchanger, in turn, generates an output signal representing *waterCondensed*, which is routed to the WRM subsystem as a source of reusable H_2O.

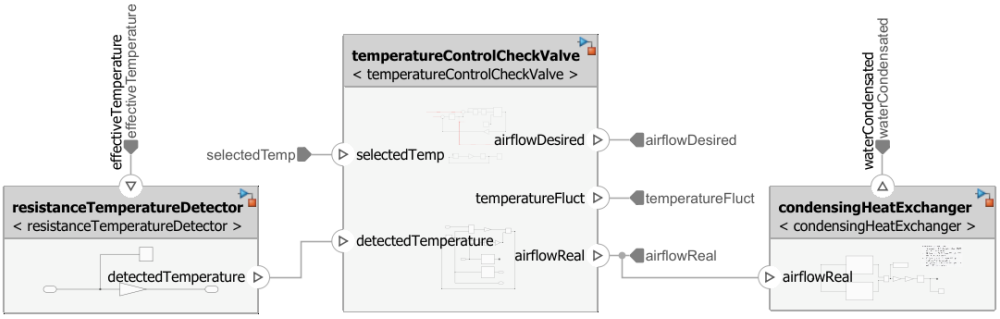


Figure 5.24: CCAA zoom in.

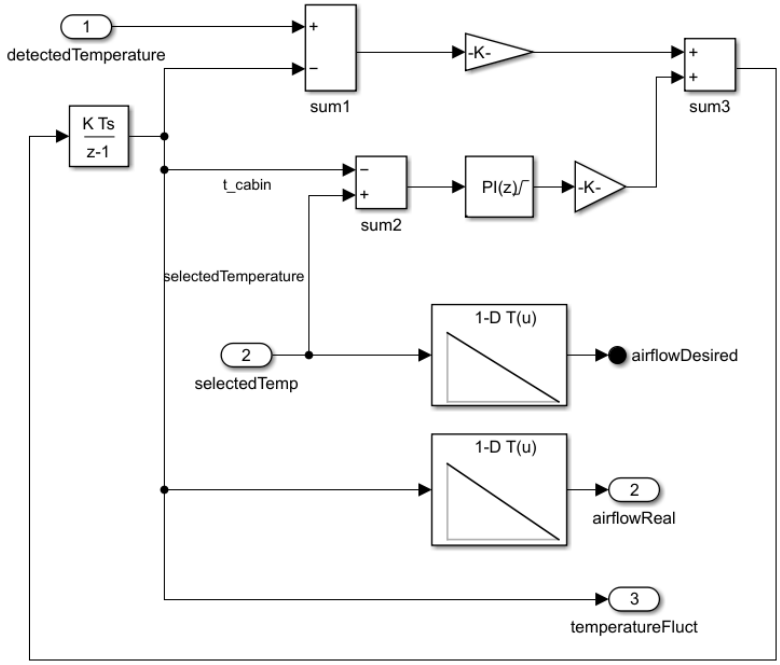


Figure 5.25: TCCV zoom in.

The *cabinAtmosphere* block provides the signals *selectedTemperature*, representing the crew's manual selection of the habitat temperature, and *effectiveTemperature*, which models the actual fluctuating temperature within the habitat.
A *Sine Wave Generator* was used to simulate the periodic 90-minute temperature cycle of the ISS cabin, with example values of 30°C as the maximum and 10 °C as the minimum, consistent with the operational range achievable by the station's TCS (Figure 5.27).

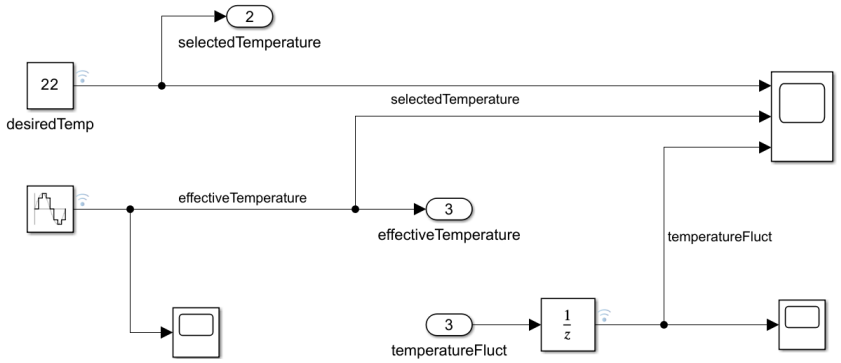


Figure 5.26: Cabin Atmosphere signal generation zoom in.

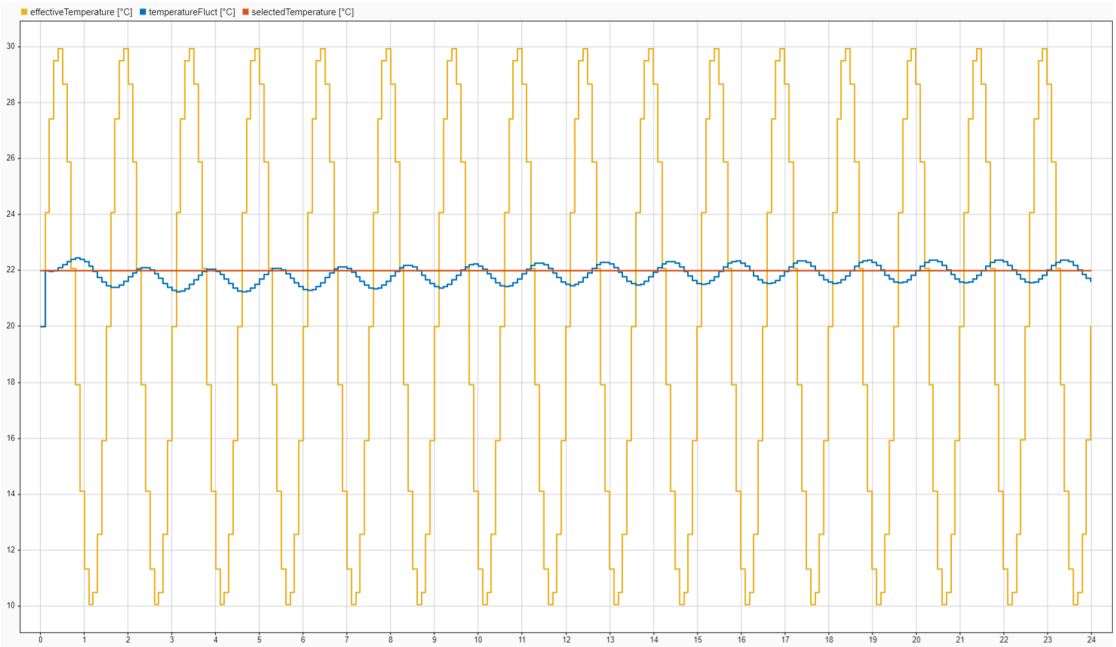


Figure 5.27: EffectiveTemperature, TemperatureFluct, and SelectedTemperature signals temporal evolution (2 crew members, 24 hours).

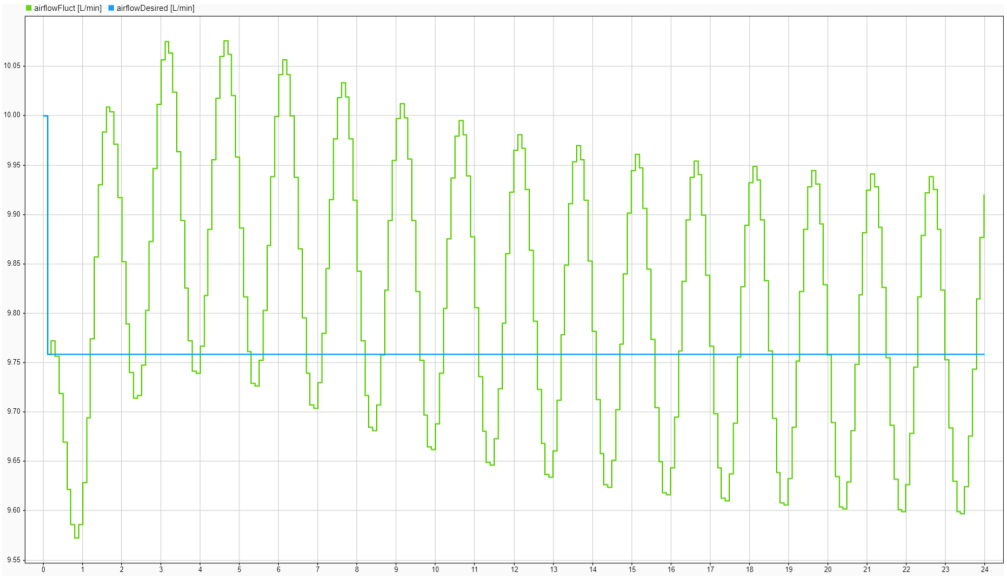


Figure 5.28: airflowFluct, and airflowDesired signals temporal evolution (2 crew members, 24 hours).

As for the ACS, Figure 5.29 presents a set of coherent requirements for the THC subsystem.

2	R-SYS-THC	Container	**THC Requirements**	⇦Implemented by: temperatureHumidityControl
2.1	R-SYS-THC-010	Container	**Control Atmospheric Temperature**	
2.1.1	R-SYS-THC-011	Functional	**Monitor Atmospheric Temperature** The Resistance Temperature Detector (RTD) shall monitor the temperature over the range of 15.6 to 32.2 °C.	
2.1.2	R-SYS-THC-012	Functional	**Remove Atmospheric Heat** The THC shall modulate the atmosphere flowrate from 8.490 to 14.150 L/min.	
2.1.3	R-SYS-THC-013	Functional	**Performance** The temperature shall be within +- 1°C (TBC) of the selected temperature.	
2.1.4	R-SYS-THC-014	Functional	**Remove Airborne Particulate Contaminants** HEPA filters shall remove 99.97% (TBC) of particulate and microorganisms.	
2.2	R-SYS-THC-020	Container	**Control Atmospheric Relative Humidity**	
2.2.1	R-SYS-THC-021	Functional	**Modulate Atmospheric Relative Humidity** The THC shall modulate the atmosphere humidity in the range from 25% to 70%.	

Figure 5.29: Example of THC Requirements.

AR Subsystem

Working in conjunction with the THC, the AR subsystem is responsible for maintaining a breathable atmosphere for the inhabitants by removing and reducing CO_2, generating oxygen, eliminating harmful contaminants, and monitoring the composition of atmospheric constituents.

Following the schematisation proposed by Wieland [37], illustrated in Figure 5.30, interactions among the various subsystems were implemented through request and production signals. Interfaces are therefore defined with the *crew* block, the *atmosphereControlSupply* block, and, for specific production flows, with the *waterRecoveryManagement* block.

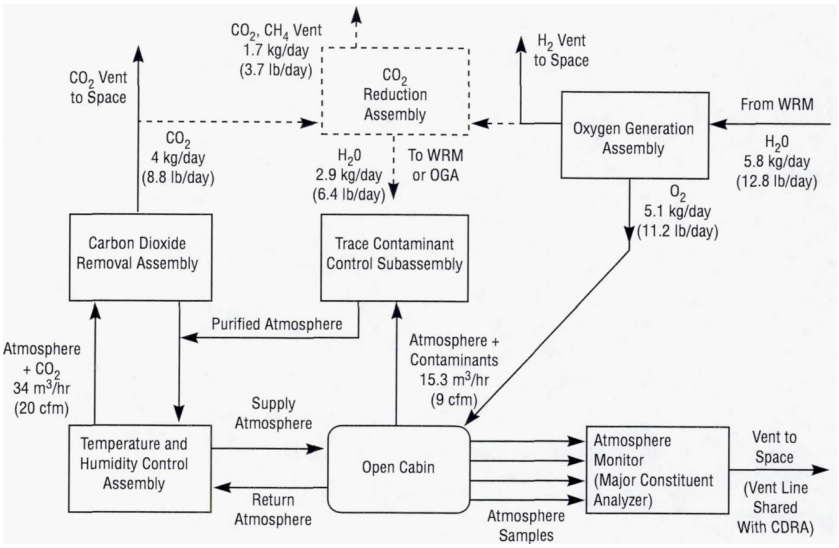


Figure 5.30: Wieland AR specifications [37].

Within the System Composer architecture, three main functional blocks can be identified: an Oxygen Generator Assembly (OGA), which models an electrolyser for the production of oxygen and hydrogen (Figure 5.32); a Carbon Dioxide Removal Assembly (CDRA), representing the system responsible for the absorption and removal of CO_2 from the cabin atmosphere; and a CO_2 Reduction Assembly, which is modelled to reduce carbon dioxide by reacting it with hydrogen to produce water and methane, which can be vented overboard or partially reintegrated into the resource loop (Figure 5.33).

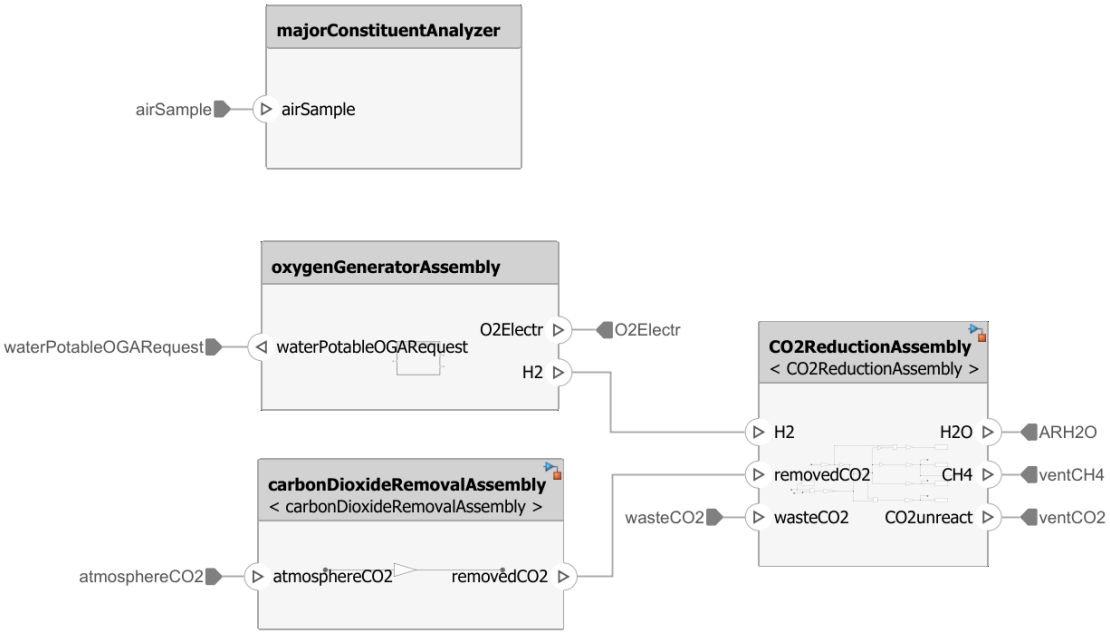


Figure 5.31: AR within System Composer

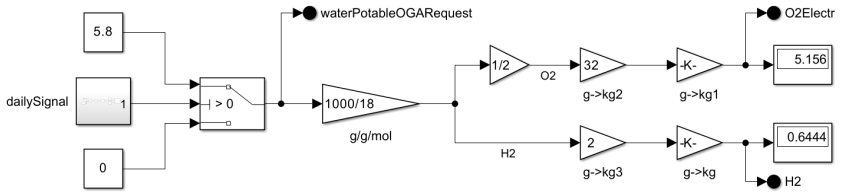


Figure 5.32: OGA Electrolyser zoom in.

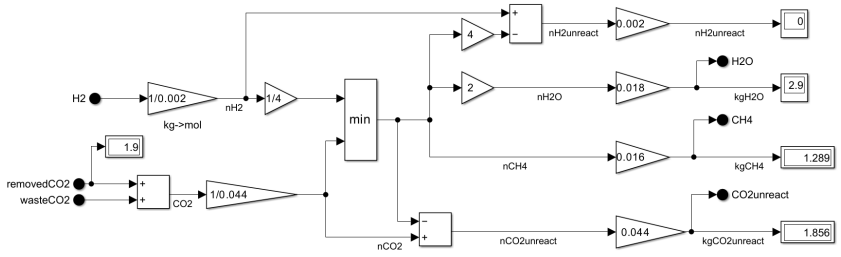


Figure 5.33: CO_2 Reduction Assembly zoom in.

As for the previous subsystems, Figure 5.34 presents an exemplary set of requirements allocated for the AR subsystem.

4	R-SYS-AR	Container	**AR Requirements**	⇦Implemented by: atmosphereRevitalisation
4.1	R-SYS-AR-010	Container	**OGA Generation**	
4.1.1	R-SYS-AR-011	Functional	**OGA O2 Generation** The OGA (Oxygen Generator Assembly) shall generate O2 up to 5.16 kg/day from 5.8 kg of H2O.	
4.1.2	R-SYS-AR-012	Functional	**OGA H2 Generation** The OGA (Oxygen Generator Assembly) shall generate H2 up to 0.64 kg/day from 5.8 kg of H2O.	
4.2	R-SYS-AR-020	Container	**CDRA**	
4.2.1	R-SYS-AR-021	Functional	**CDRA Power** The CDRA shall need a 587W (TBC) power supply.	
4.3	R-SYS-AR-030	Container	**MCA**	
4.3.1	R-SYS-AR-031	Functional	**MCA Detection** The MCA shall detect partial pressure of N2, O2, H2, CH4, and CO2.	
4.3.2	R-SYS-AR-032	Functional	**N2 accuracy** The MCA shall detect N2 partial pressures with an accuracy of 2%.	
4.3.3	R-SYS-AR-033	Functional	**O2 accuracy** The MCA shall detect O2 partial pressures with an accuracy of 2%.	
4.3.4	R-SYS-AR-034	Functional	**H2 accuracy** The MCA shall detect H2 partial pressures with an accuracy of 5%.	
4.3.5	R-SYS-AR-035	Functional	**CH4 accuracy** The MCA shall detect CH4 partial pressures with an accuracy of 5%.	
4.3.6	R-SYS-AR-036	Functional	**CO2 accuracy** The MCA shall detect CO2 partial pressures with an accuracy of 1%.	

Figure 5.34: Example of AR Requirements.

WM Subsystem

The WM subsystem is designed to collect, process, and store the various waste streams generated within a space habitat. These include metabolic waste, other solid residues, and both liquid and gaseous effluents. For long-duration missions, it is necessary to employ recovery techniques that extract useful by-products from these wastes, whereas shorter missions may rely on storage without extensive processing.

For this didactic study, the waste collection and storage functions were divided between the *wasteManagement* and *storageFacilities* blocks in order to preserve linearity in the architectural modelling.
As no internal functional complexity was modelled, this subsystem acts primarily as a logical junction within the architecture, receiving *urine* and *faeces* flows and routing them both to the storage facilities and to the WRM subsystem.

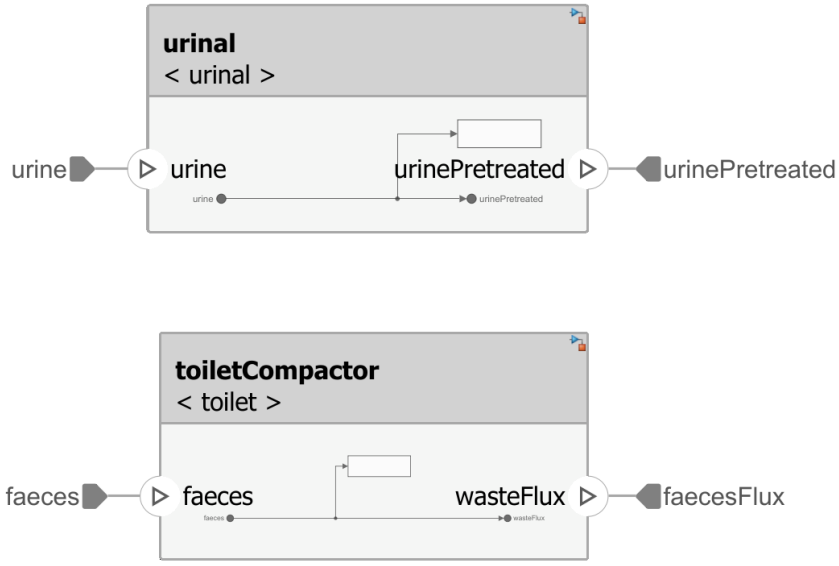


Figure 5.35: WM within System Composer.

WRM Subsystem

The WRM subsystem provides water for crew hygiene, consumption, and other operational activities. For short-duration missions, water can be stored in tanks; however, the required volume becomes prohibitive for longer missions. Given the high purity standards required for human consumption, potable water may be supplied either by fuel cells[2] or through waste water purification.

For this subsystem, following the guidelines provided by Wieland [37], two architectural blocks were defined, as shown in Figure 5.36. These are the Urine Processor (UP) and the Water Processor (WP). The former is responsible for processing urine into solid waste and wastewater, while the latter stores and processes the wastewater produced by the UP, closing the loop by generating potable water for the entire system.

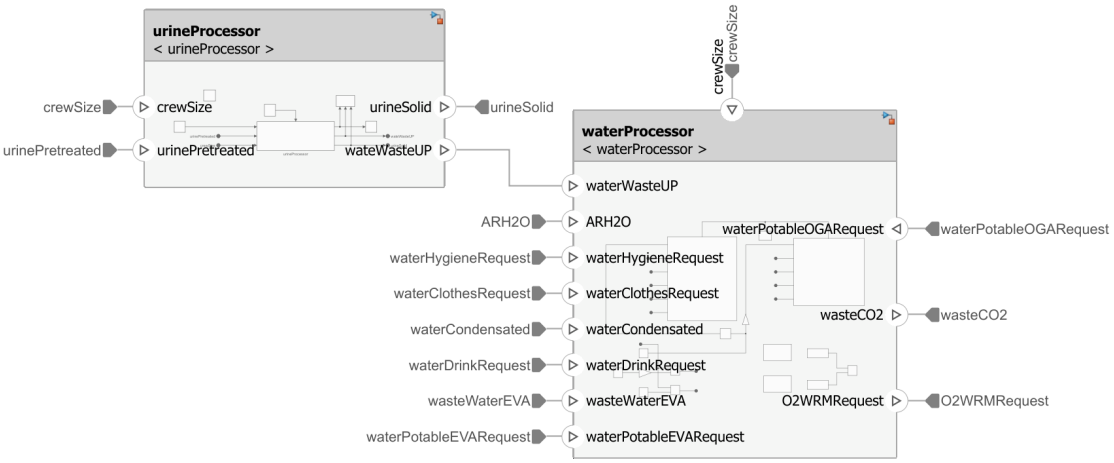


Figure 5.36: WRM within System Composer.

[2] H_2 and O_2 are electrochemically combined, generating electricity and producing water as a by-product.

As for the previous subsystems, an impulsive logic was adopted for the WP. Through the use of a *Pulse Generator*, this approach enables the activation of a triggered subsystem (Figure 5.38) that processes a defined quantity of flow within the loop. This design choice introduces a discretisation of the wastewater flow, scaled according to the number of crew members aboard the habitat.

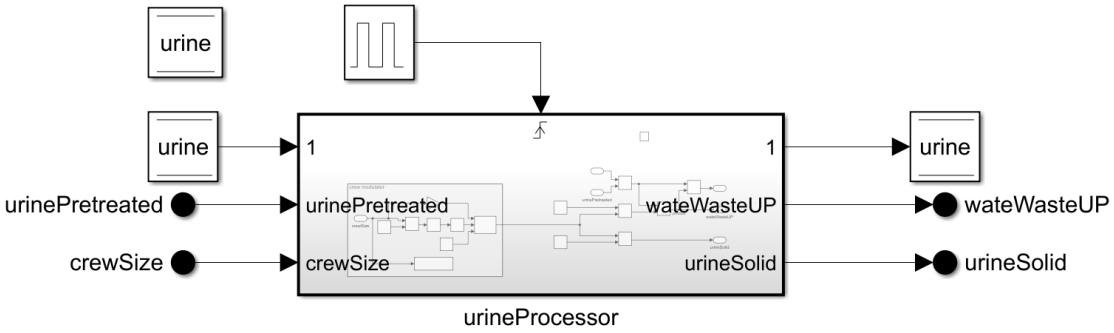


Figure 5.37: UP zoom in.

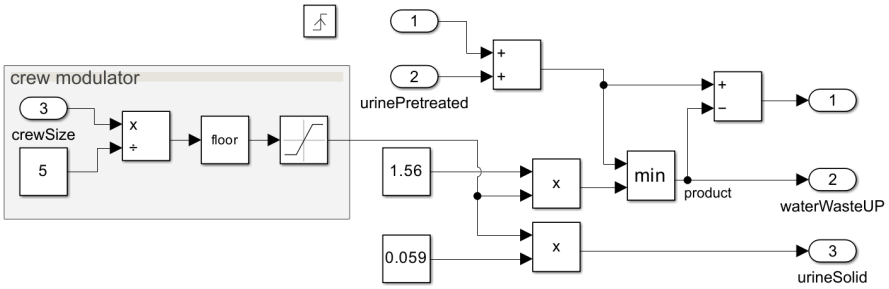


Figure 5.38: UP zoom in.

Similarly to the WP, as shown in Figure 5.39, two triggered subsystems were implemented. These subsystems perform a straightforward aggregation or depletion of the incoming flows, depending on whether they represent system demands (e.g. *waterPotableOGARequest*, *waterHygieneRequest*, etc.) or incoming productions (e.g. *waterCondensated*, *waterWasteUP*, etc.).

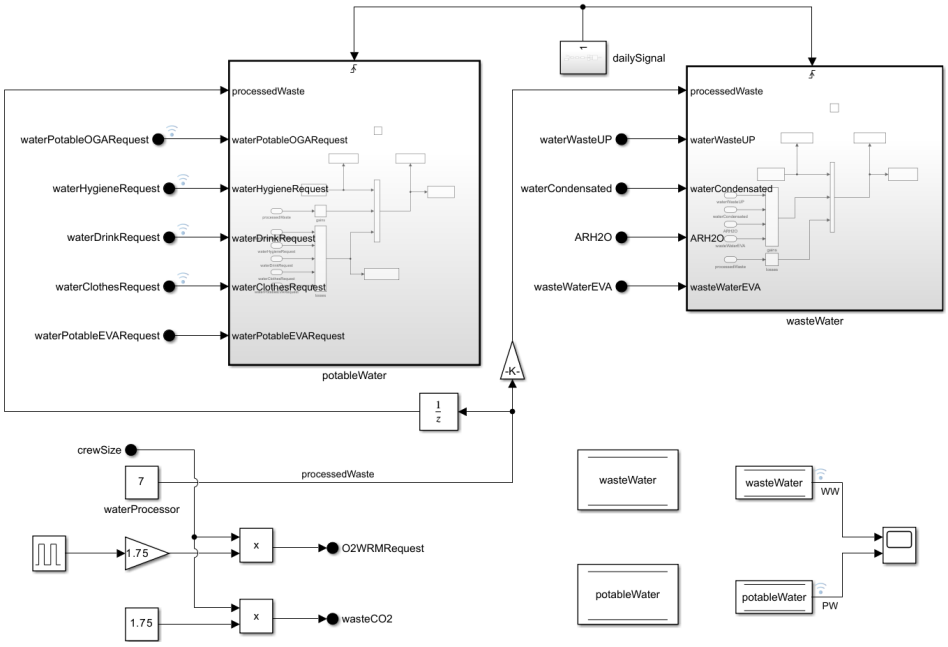


Figure 5.39: WP zoom in.

Figure 5.40 illustrates the trends in wastewater accumulation and potable water depletion as a function of varying crew-size scenarios, highlighting the sensitivity of the system to changes in human payload.
This is a clear example of how complex it can be to properly close the internal loops of a space habitat, given the presence of multiple interacting variables within the system.

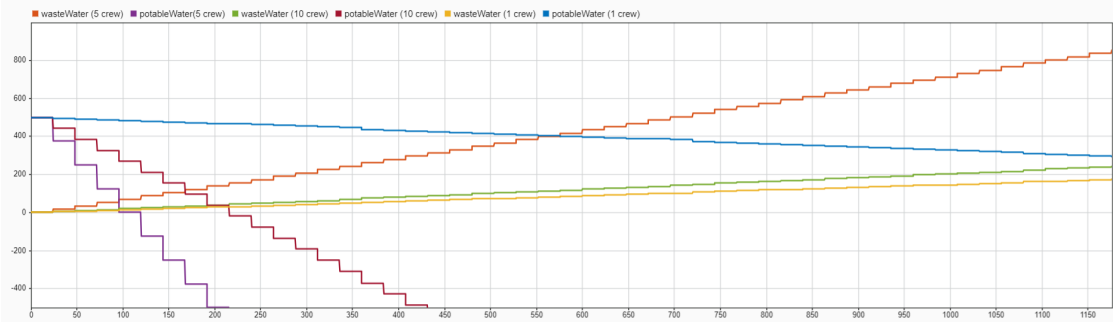


Figure 5.40: wasteWater, and potableWater signals temporal evolution (1/5/10 crew member, 25 days)

As for the previous subsystems, Figure 5.41 presents a brief set of plausible, WRM-related requirements.

6	R-SYS-WRM	Container	**WRM Requirements**	⇦Implemented by: waterRecoveryManagement
6.1	R-SYS-WRM-010	Container	**WP performance**	
6.1.1	R-SYS-WRM-011	Functional	**WP wastewater processing** The WP shall process 9.2 to 13.6 kg/day/ person of wastewater.	
6.1.2	R-SYS-WRM-012	Container	**WP potable water providing**	
6.1.2.1	R-SYS-WRM-0121	Functional	**Food rehydration** The WP shall provide 2.8 kg/day/person of water for food rehydration.	
6.1.2.2	R-SYS-WRM-0122	Functional	**Hygiene** The WP shall provide 6.8 kg/day/person of water for hygiene.	
6.1.2.3	R-SYS-WRM-0123	Functional	**Payload use** The WP shall provide 2.2 kg/day/person of water for payload use.	
6.2	R-SYS-WRM-020	Container	**UP Performance**	
6.2.1	R-SYS-WRM-021	Functional	**UP urine processing** The UP shall process 1.56 kg/person/day of urine for up to 6 crew members.	

Figure 5.41: Example of WRM Requirements

FDS Subsystem

Fire represents one of the most serious hazards within any space habitat. As mission durations increase, so too does the likelihood of a fire, making it essential to minimise ignition sources and reduce overall risk. The FDS subsystem must identify any outbreak at the earliest possible stage, typically through smoke or flame detection technologies. Should a fire occur, suitable suppression measures must be available to extinguish it effectively. Combustion inevitably produces by-products that may be harmful to the crew, so it is critical that these contaminants are promptly removed from the cabin atmosphere.

For this subsystem, no internal architectural components were explicitly modelled, due to the extensive scope of the subsystem, which would have required a level of detail significantly higher than that adopted for the previously described subsystems. Nevertheless, based on the documentation provided in [37], it was possible to derive a set of plausible requirements that describe and formalise the main functions and responsibilities of this subsystem, as shown in Figure 5.42.

5	R-SYS-FDS	Container	**FDS Requirements**	Implemented by: fireDetectionSuppression
5.1	R-SYS-FDS-010	Functional	**Zone Isolation** The FDS shall isolate the fire within 30 sec.	
5.2	R-SYS-FDS-020	Container	**Extinguish Fire**	
5.2.1	R-SYS-FDS-021	Functional	**Fire Suppressant** The FDS shall apply a fire suppressant at each potential fire source location.	
5.2.2	R-SYS-FDS-021	Functional	**Atmosphere Vent** The FDS shall vent the atmosphere to achieve an O2 concetration of <3.4 kPa within 10 min.	

Figure 5.42: Example of FDS Requirements.

EVA Support

During extravehicular activity, astronauts transition from the spacecraft's high-pressure cabin to the substantially lower pressure of the spacesuit. This shift introduces a significant risk: nitrogen dissolved in body tissues and the bloodstream can come out of solution and form bubbles, leading to decompression sickness. To mitigate this hazard, crew members breathe pure oxygen in a hyperbaric environment prior to undertaking an EVA. This procedure promotes the removal of dissolved nitrogen from the body with each exhalation, thereby reducing the likelihood of bubble formation. The Environmental Control and Life Support System must therefore enable EVA activities by maintaining the environmental conditions required for their safe execution.

In this last subsystem, a simplified interaction between the *EVA* block and the *waterRecoveryManagement* block has been modelled, with the sole purpose of representing the demand for potable water associated with extravehicular activities and the corresponding production of *wastewater* to be reintegrated into the water recovery loop.
The use of a *Constant* block allows the number of simultaneously active *space suits* to be easily parameterised, enabling the simple assessment of the impact of EVA operations on the system water balance.
A further refinement of the model could include an explicit representation of the oxygen consumption associated with EVA activities, together with the corresponding production of carbon dioxide, thereby enabling a more consistent closure of the atmospheric mass balance within the system.

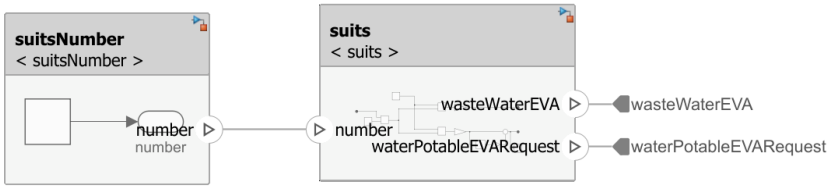


Figure 5.43: EVA within System Composer.

Figure 5.44 presents a representative set of requirements associated with EVA activities.

7	R-SYS-EVA	Container	**EVA Requirements**	⇦Implemented by: EVA
7.1	R-SYS-EVA-010	Functional	**Space suits pressure** The space suits shall operate at 29.6 kPa.	
7.2	R-SYS-EVA-020	Container	**Support servicies**	
7.2.1	R-SYS-EVA-021	Functional	**Oxygen support** The ECLSS shall provide TBD kg of O2 for each suits.	
7.2.2	R-SYS-EVA-022	Functional	The ECLSS shall provide TBD kg of N2 for each suits.	
7.2.3	R-SYS-EVA-023	Functional	**Water support** The ECLSS shall provide TBD kg of H2O for each suits.	

Figure 5.44: Example of EVA Requirements.

Conclusions and Future works

This thesis set out to investigate the potential of System Composer as a Model-Based Systems Engineering (MBSE) tool for space system conceptual design, with particular emphasis on its applicability in academic, didactic, and early-stage engineering contexts. The work pursued three main objectives: to evaluate System Composer in comparison with other MBSE tools currently available; to develop a conceptual design of a CubeSat mission using MBSE principles in order to assess the tool's capability to standardise system architectures across different levels of abstraction; and to demonstrate the strengths of the tool through the digital modelling of the International Space Station(ISS) Environmental Control and Life Support System (ECLSS), explicitly integrating the human being as the central payload through Human-Centred Engineering principles.

The results obtained show that System Composer represents a particularly accessible MBSE tool within academic environments, largely due to its integration within the MATLAB/Simulink ecosystem, which is already widely adopted in engineering education and research. This familiarity significantly lowers the entry barrier for students and researchers approaching MBSE for the first time, facilitating both learning and practical application without the need for extensive additional training or the adoption of entirely new software environments.

From a modelling perspective, the block-based architecture of System Composer proved effective in supporting the design of both small-scale and large-scale system architectures. The ability to represent logical, functional, and physical relationships through modular blocks enables a high degree of flexibility and dynamism, allowing multiple architectures and design alternatives to be explored in parallel. This characteristic was shown to be particularly advantageous when working across different levels of detail and abstraction, supporting iterative refinement without compromising model coherence.

A significant strength identified in this work is the presence of an internal requirements management editor. This feature simplifies the definition, modification, and allocation of requirements directly within the architectural model, reducing the reliance on external tools that may otherwise fragment the MBSE process. The effectiveness of this capability is further enhanced by the possibility of implementing internal verification and validation activities through the same block-based modelling paradigm, resulting in an integration of requirements traceability and V&V activities into a unified modelling environment, avoiding the need for additional formal languages or specialised verification tools in the early design phases.

The application of MBSE principles to both unmanned and manned systems demonstrated the methodological versatility of the approach. In particular, the modelling of the ISS ECLSS highlighted how the human being can be explicitly introduced into the architectural framework as a payload with specific needs and constraints. This case study showed that MBSE, when supported by System Composer, can accommodate Human-Centred Engineering principles, enabling a structured representation of interactions between technical subsystems and human requirements within a single, coherent model.

While System Composer cannot address all possible MBSE needs, the findings of this thesis suggest that it offers a well-balanced compromise between modelling flexibility, usability, and learning curve. These characteristics make it especially suitable for educational and research-oriented environments, as well as for early-stage conceptual design activities where rapid iteration and architectural exploration are essential.

Ongoing and future developments, such as the integration of external plug-ins and interoperability with other MBSE tools (for example, Cameo), indicate promising directions for extending System Composer's capabilities, particularly in areas such as project-level timeline management and advanced system-level analyses. These developments provide a strong motivation for continued research into the use of System Composer within the broader MBSE framework. Future work may include the application of the tool to additional case studies, both to expand its educational value and to further assess its suitability for industrial and operational design scenarios.

Appendix A

Functions

A.1 findComponentRecursive.m

This function performs a recursive search within a System Composer architecture to locate a component by its name. It begins by retrieving all components contained in the input architecture (arch.Components) and iteratively compares each component's name with the specified targetName. If a match is found, the corresponding component is returned. If not, the function checks whether the current component contains a sub-architecture. In such cases, the function calls itself recursively to continue the search within nested architectures. The process terminates once the target component is found or all hierarchical levels have been explored.

```
1  function comp = findComponentRecursive(arch, targetName)
2      comps = arch.Components;
3      comp = [];
4      for i = 1:length(comps)
5          if comps(i).Name == targetName
6              comp = comps(i);
7              return
8          end
9          if ~isempty(comps(i).Architecture)
10             comp = findComponentRecursive(comps(i).Architecture,
    targetName);
11             if ~isempty(comp)
12                 return
13             end
14         end
15     end
16 end
```

A.2 fnc_budget.m

The fnc_budget function performs a systematic extraction and aggregation of specific property values, such as mass or power, from the components of a System Composer model named SatelliteModel. It opens the model, retrieves the root architecture, and iterates through a predefined list of components. For each component, it locates the corresponding architectural element, reads the property value associated with the specified type (e.g., mass), and assigns this value to a variable in the MATLAB base workspace. Finally, the function computes and stores the total sum of all component values, enabling efficient budget evaluation across the entire system.

Table A.1: Types list for budgeting.

Type	Description
Mass	Reads and saves the mass properties in kg
Cost	Reads and saves the cost properties in M€
Volume	Reads and saves the volume properties in the format $l \times l \times l$ in m

```
function fnc_budget(type)
    archModel = systemcomposer.openModel('SatelliteModel');
    rootArch = archModel.Architecture;

    compNames={"Payload","TCS","CnDH","comSys","EPS","AODCS","
   Propulsion","Structure"};
    compProp={"Payload","Thermal","CommandDataHandling","
   Communication","Power","GuideNavCntrl","Propulsion","Structure
   "};

    func=zeros(size(compNames));

    for i = 1:numel(compNames)

        comp = findComponentRecursive(rootArch, compNames{i});
        val = getPropertyValue(comp, sprintf('%s.%s.%s','
   SatelliteProfile',compProp{i},type));

        val = strrep(val, '''', '');

        if strcmpi(type,'volume')
            dims = str2double(split(strrep(val,' ',''),'x'));

            if numel(dims) == 3 && all(~isnan(dims))
                func(i) = prod(dims);
```

```
22              end
23          else
24              func(i) = str2double(val);
25          end
26          assignin('base',sprintf('%s_%s',type,compNames{i}),func(i)
    );
27      end
28
29      assignin('base',sprintf('%s_%s',type,'total'),sum(func));
30  end
```

A.3 fnc_modes.m

Table A.2: Cmd list.

Cmd	Description
sumModes	Provide a graph with the sum of the various consumpions for each Operative modes.
fullModes	Provide a matrix with the expected consumption for each Operative modes along a graphical rappresentation.

```
1  function [matrix, vectorPow,vectorTime] = fnc_modes(cmd)
2      % Open the model architecture
3      archModel = systemcomposer.openModel('SatelliteModel');
4      rootArch = archModel.Architecture;
5
6      % Assign the to-find components and properties
7      compNames = {"Dormant","Detumbling", "Commissioning", "
     OrbitInsertion","Nominal","Manouvering","PayModeGround","
     PayModeCity","Transmission","SafeMode","Disposal"};
8      propNames = {"LWIR", "VNIR", "PCDU","tranceiver","transmitter
     ","antennas","OBC","starTracker1","starTracker2","sunSensors","
     reactionWheels","IMU","magnetorquers","GNSS","PCU","thrusters
     ","tempSensors"};
9
10     % Initialise a blanck matix
11     matrix = zeros(numel(compNames), numel(propNames));
12     vectorTime = zeros(1,numel(compNames));
13
14     % Recursively search for the combinations of components and
     properties
15     for i = 1:numel(compNames)
```

```
16          compName = findComponentRecursive(rootArch, compNames{i});
17          vectorTime(i) = str2double(getPropertyValue(compName,'
    SatelliteProfile.operativeModes.modeDuration'));
18
19          for j = 1:numel(propNames)
20              name=sprintf('%s.%s','SatelliteProfile.operativeModes'
    , propNames{j});
21              val = getPropertyValue(compName, name);
22              compProp = findComponentRecursive(rootArch, propNames{
    j});
23
24                  if strcmpi(val, "'on'")
25                      matrix(i,j) = str2double(getPropertyValue(
    compProp,'SatelliteProfile.BaseComponent.powerConsOn'));
26                  elseif strcmpi(val, "'off'")
27                      matrix(i,j) = 0;
28                  elseif strcmpi(val, "'stand_by'")
29                      matrix(i,j) = str2double(getPropertyValue(
    compProp,'SatelliteProfile.BaseComponent.powerConsStb'));
30                  elseif strcmpi(val,"'peak'")
31                      matrix(i,j) = str2double(getPropertyValue(
    compProp,'SatelliteProfile.BaseComponent.powerConsPeak'));
32                  else
33                      % se la proprietà è altro tipo, prova numerico
34                      matrix(i,j) = str2double(val);
35                  end
36          end
37      end
38
39      % Shows the labes
40      compNames = ["Dormant" "Detumbling" "Commissioning" "
    OrbitInsertion" "Nominal" "Manouvering" "PayModeGround" "
    PayModeCity" "Transmission" "SafeMode" "Disposal"];
41      propNames = ["LWIR", "VNIR", "PCDU","tranceiver","transmitter
    ","antennas","OBC","starTracker1","starTracker2","sunSensors","
    reactionWheels","IMU","magnetorquers","GNSS","PCU","thrusters
    ","tempSensors"];
42
43      vectorPow=sum(matrix,2);
44
45      if strcmpi(cmd,'sumModes')
46          figure(1)
47          h=bar3h(vectorPow);
48          zlabel('Subsystem');
49          ylabel('Power [W]');
50          title('Power Profile per Mode');
51          set(gca, 'ZTick', 1:numel(compNames), 'ZTickLabel',
    compNames);
52          clim([min(vectorPow) max(vectorPow)]);
```

```
53          for k = 1:length(h)
54              zdata = h(k).YData;
55              h(k).CData = zdata;
56              h(k).FaceColor = 'interp';
57          end
58          colormap(parula);
59          colorbar;
60      elseif strcmpi(cmd,'fullModes')
61          figure(1)
62          h=bar3(matrix);
63          xlabel('Component');
64          ylabel('Operative Mode');
65          zlabel('Power [W]');
66          title('Power Profile per Component per Operative mode');
67          set(gca, 'XTick', 1:numel(propNames), 'XTickLabel',
    propNames);
68          set(gca, 'YTick', 1:numel(compNames), 'YTickLabel',
    compNames);
69          for k = 1:length(h)
70              zdata = h(k).ZData;
71              h(k).CData = zdata;
72              h(k).FaceColor = 'interp';
73          end
74          colormap(parula);
75          colorbar;
76          matrix = array2table(matrix, 'VariableNames', propNames, '
    RowNames', compNames);
77      end
78
79  end
```

A.4 findComponentParent().m

```
1   function parentName = findComponentParent(arch, targetName)
2       parentName = [];
3
4       for i = 1:numel(arch.Components)
5           comp = arch.Components(i);
6
7           if strcmp(comp.Name, targetName)
8               parentName = string(arch.Name);
9               return;
10          end
11
12          if ~isempty(comp.Architecture)
```

```
13              parentName = findComponentParent(comp.Architecture,
    targetName);
14              if ~isempty(parentName)
15                  return;
16              end
17          end
18      end
19  end
```

A.5 fnc_powerProfile().m

```
1  function [matrix, vectorOn] = powerProfile()
2      archModel = systemcomposer.openModel('SatelliteModel');
3      rootArch = archModel.Architecture;
4
5      sysNames = {"Payload","CnDH","comSys","TCS","EPS","AODCS","
     Propulsion"};
6      propNames = {"LWIR", "VNIR", "PCDU","tranceiver","transmitter"
     ,"antennas","OBC","starTracker1","starTracker2","sunSensors","
     reactionWheels","IMU","magnetorquers","GNSS","PCU","thrusters",
     "tempSensors"};
7      matrix = zeros(3, numel(propNames));
8      vectorOn = zeros(1,numel(sysNames));
9
10     for i = 1:numel(propNames)
11         comp = findComponentRecursive(rootArch,propNames{i});
12         matrix(1,i) = str2double(getPropertyValue(comp,'
     SatelliteProfile.BaseComponent.powerConsStb'));
13         matrix(2,i) = str2double(getPropertyValue(comp,'
     SatelliteProfile.BaseComponent.powerConsOn'))*1.05;
14         for j = 1:numel(sysNames)
15             if strcmpi(sysNames{j},findComponentParent(rootArch,
     propNames{i}))
16                 vectorOn(j) = vectorOn(j)+matrix(2,i);
17             end
18         end
19         matrix(3,i) = str2double(getPropertyValue(comp,'
     SatelliteProfile.BaseComponent.powerConsPeak'));
20     end
21
22     propNames = ["LWIR" "VNIR" "PCDU" "tranceiver" "transmitter" "
     antennas" "OBC" "starTracker1" "starTracker2" "sunSensors" "
     reactionWheels" "IMU" "magnetorquers" "GNSS" "PCU" "thrusters"
     "tempSensors"];
23     consNames = ["Stand-by","On","Peak"];
24     figure(1)
```

```
25      h=bar3(matrix);
26      xlabel('Component');
27      ylabel('Consumption Mode');
28      zlabel('Power [W]');
29      title('Power Profile per Component per Consumption mode');
30      set(gca, 'XTick', 1:numel(propNames), 'XTickLabel', propNames)
        ;
31      set(gca, 'YTick', 1:numel(consNames), 'YTickLabel', consNames)
        ;
32      for k = 1:length(h)
33          zdata = h(k).ZData;
34          h(k).CData = zdata;
35          h(k).FaceColor = 'interp';
36      end
37      colormap(parula);
38      colorbar;
39      %matrix = array2table(matrix, 'VariableNames', propNames, '
        RowNames', consNames);
40
41      sysNames = ["Payload" "CnDH" "comSys" "TCS" "EPS" "AODCS" "
        Propulsion"];
42      figure(2)
43      h=bar3(vectorOn);
44      ylabel('Subsystem');
45      zlabel('Power [W]');
46      title('Power Profile per Subsystem');
47      set(gca, 'YTick', 1:numel(sysNames), 'YTickLabel', sysNames);
48      clim([min(vectorOn) max(vectorOn)]);
49      for k = 1:length(h)
50          zdata = h(k).ZData;
51          h(k).CData = zdata;
52          h(k).FaceColor = 'interp';
53      end
54      colormap(parula);
55      colorbar;
56  end
```

Appendix B

Scripts

B.1 powerConsOrbit.m

```
clc
close all
clear

archModel = systemcomposer.openModel('SatelliteModel');
rootArch = archModel.Architecture;

modeNames = ["Dormant" "Detumbling" "Commissioning" "
    OrbitInsertion" "Nominal" "Manouvering" "PayModeGround" "
    PayModeCity" "Transmission" "SafeMode" "Disposal"];
[matrix, powerVector, durationVector] = fnc_modes(0);

vectorData = zeros(7,1);
power = zeros(3,1);
duty = zeros(3,1);
T = 5670.8/3600;

% detumbling
    power(1) = powerVector(2);
    power(2) = 0;
    power(3) = 0;

    duty(1) = 100;
    duty(2) = 0;
    duty(3) = 0;

        vectorData(1) = power(1)*duty(1)/100;

% commissioning
```

```
28      power(1) = powerVector(3);
29      power(2) = 0;
30      power(3) = 0;
31
32      duty(1) = 100;
33      duty(2) = 0;
34      duty(3) = 0;
35
36          vectorData(2) = (power(1)*duty(1)+power(2)*duty(2)+power
        (3)*duty(3))/100;
37
38  % insetion + nominal
39      power(1) = powerVector(4);
40      power(2) = powerVector(5);
41      power(3) = 0;
42
43      duty(1) = durationVector(4)/T*100;
44      duty(2) = (durationVector(5)-durationVector(4))/T*100;
45      duty(3) = 0;
46
47          vectorData(3) = (power(1)*duty(1)+power(2)*duty(2)+power
        (3)*duty(3))/100;
48
49  % city mapping + nominal + manouvering
50      power(1) = powerVector(8);
51      power(2) = powerVector(5);
52      power(3) = powerVector(6);
53
54      duty(1) = durationVector(8)/T*100;
55      duty(2) = (durationVector(5)-durationVector(8)-durationVector
        (6))/T*100;
56      duty(3) = durationVector(6)/T*100;
57
58          vectorData(4) = (power(1)*duty(1)+power(2)*duty(2)+power
        (3)*duty(3))/100;
59
60  % transmission + nominal + manouvering
61      power(1) = powerVector(9);
62      power(2) = powerVector(5);
63      power(3) = powerVector(6);
64
65      duty(1) = durationVector(9)/T*100;
66      duty(2) = (durationVector(5)-durationVector(9)-durationVector
        (6))/T*100;
67      duty(3) = durationVector(6)/T*100;
68
69          vectorData(5) = (power(1)*duty(1)+power(2)*duty(2)+power
        (3)*duty(3))/100;
70
```

```
71          % 2nd expensive duty cycle graph
72              powerGraph = [0 power(1) power(1) power(2) power(2)
    power(3) power(3) 0];
73              timeGraph = [0 0 duty(1) duty(1) (duty(1)+duty(2)) (
    duty(1)+duty(2)) 100 100];
74              avgPowerGraph = [mean(powerGraph) mean(powerGraph)];
75              avgTime = [0, 100];
76
77              figure('Position',[100, 100, 1200, 500])
78              scatter(timeGraph, powerGraph, 'Visible', 'off');
79              hold on;
80              plot(timeGraph, powerGraph, '-r','Color',[0.8500
    0.3250 0.0980], 'LineWidth', 1.5);
81
82              scatter(avgTime,avgPowerGraph,'Visible', 'off');
83              plot(avgTime, avgPowerGraph,'--','Color',[0 0.4470
    0.7410], 'LineWidth', 1.5);
84
85              scatter(avgTime,avgPowerGraph*1.2,'Visible', 'off');
86              plot( avgTime,avgPowerGraph*1.2, '-.g','Color',[0.4660
     0.6740 0.1880], 'LineWidth', 1.5);
87              xlabel('dutyCicle [%]');
88              %ylabel('Power [W]');
89              title('Power budget of Transmission-Nominal-
    Manouvering modes combination');
90              legend('','Power [W]','','Average Power [W]','','
    Average Power w/ margin [W]');
91              grid on;
92
93
94  % ground mapping + nominal + manouvering
95      power(1) = powerVector(7);
96      power(2) = powerVector(5);
97      power(3) = powerVector(6);
98
99      duty(1) = durationVector(7)/T*100;
100     duty(2) = (durationVector(5)-durationVector(7)-durationVector
    (6))/T*100;
101     duty(3) = durationVector(6)/T*100;
102
103         vectorData(6) = (power(1)*duty(1)+power(2)*duty(2)+power
    (3)*duty(3))/100;
104
105 % safe
106     power(1) = powerVector(10);
107     power(2) = 0;
108     power(3) = 0;
109
110     duty(1) = durationVector(10)/T*100;
```

```
111     duty(2) = 0;
112     duty(3) = 0;
113 
114         vectorData(7) = (power(1)*duty(1)+power(2)*duty(2)+power
    (3)*duty(3))/100;
115 
116 namesCombinations = ["Detumbling","Commissioning","OrbitInsertion-
    Nominal","PayModeCity-Nominal-Manouvering","Transmission-
    Nominal-Manouvering","PayModeGround-Nominal-Manouvering","
    SafeMode"];
117 figure('Position',[100, 100, 800, 500])
118     h=bar3h(vectorData);
119     zlabel('Modes combinations');
120     ylabel('Power [W]');
121     title('Power Consumption of modes combinations');
122     set(gca, 'ZTick', 1:numel(namesCombinations), 'ZTickLabel',
    namesCombinations);
123     clim([0 max(vectorData)]);
124     for k = 1:length(h)
125         zdata = h(k).YData;
126         h(k).CData = zdata;
127         h(k).FaceColor = 'interp';
128     end
129     colormap(parula);
130     colorbar;
131 
132 %for i = 1:numel(modeNames)
133 %     fprintf('%s power = %d\n',modeNames(i),sat.(modeNames(i)).
    power);
134 %end
135 %fprintf('\n');
136 %for i = 1:numel(modeNames)
137 %     fprintf('%s duration = %d\n',modeNames(i),sat.(modeNames(i)).
    duration);
138 %end
```

Appendix C

Spaceflight

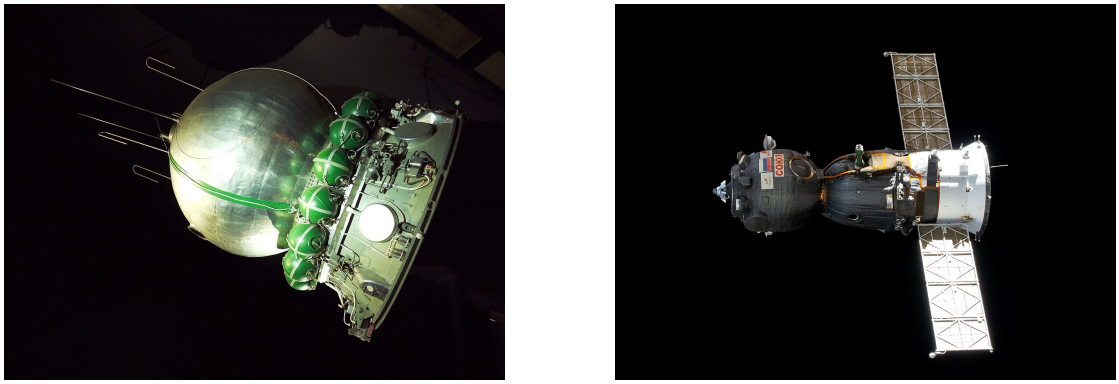
C.1 Human Spaceflight

The beginning of human spaceflight can be traced to 12 April 1961, when Yuri Gagarin completed a single orbit around the Earth aboard the Vostok-1 spacecraft. Over the following fifty years, more than five hundred individuals have travelled beyond an altitude of 100 kilometres. Depending on their country of origin, these explorers are referred to as astronauts, cosmonauts, or taïkonauts.
During this period, only three nations have launched humans into space: the Soviet Union/Russia, the United States, and China. Each nation adopted a progressive approach, building successive missions upon the technical and operational achievements of their predecessors and capitalising on lessons learned along the way.
Today, international cooperation defines human spaceflight. The International Space Station (ISS), jointly developed by the United States, Russia, Europe, Canada, and Japan, has been continuously inhabited for more than thirteen years. Meanwhile, China launched Tiangong-1 in 2011, a human-tended orbital laboratory that laid the groundwork for its current modular space station programme [48].

C.1.1 Soviet Union and Russia's programme

The first human spaceflight programme was accomplished by the Soviet Union with the Vostok programme (1961-1963, Figure C.1a), marking the first time a human completed an orbital flight around the Earth while also spending a complete day in space. In the following years, the Voskhod programme (1964-1965) conducted two missions with multi-person crews, performing the first extravehicular activity (EVA) in human history. Soviet human spaceflight culminated in 1967 with the introduction of the Soyuz spacecraft (Figure C.1b), which remains in operation

for missions to the International Space Station (ISS). The Soyuz supported the operation of both the Salyut (1971–1986) and Mir (1986–2001, Figure C.2) space stations, the former being a single-module platform and the latter a multi-module orbital complex [49].



(a) **(b)**

Figure C.1: On the left, Vostok-1 [50] and, on the right, Soyuz MS [51] spacecrafts.

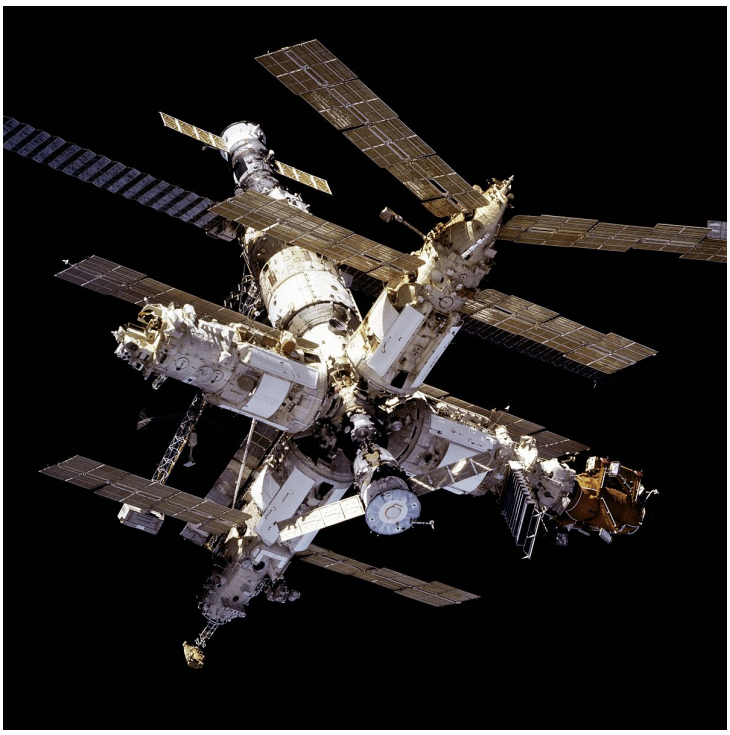


Figure C.2: Mir space station [52].

C.1.2 United States programme

U.S. began their human spaceflight with the Mercury programme (1961-1963, Figure C.3a), which included 25 missions with the objectives of placing a human spacecraft into Earth orbit, evaluating the astronaut's ability to operate in space, and ensuring the safe recovery of both the crew and the vehicle.
The successor to Mercury was the Gemini programme (1965-1966, Figure C.3b), designed to prepare for future lunar missions by testing long-duration flights, performing orbital rendezvous and docking, refining re-entry techniques, and studying the physiological impacts of extended spaceflight on humans.
The largest programme was Apollo (1961-1972, Figure C.4a), which accomplished successful lunar landings and safe returns to Earth, while contributing to technological advancements in support of national objectives in space, enabling the scientific exploration of the Moon, and enhancing humanity's ability to operate in the extraterrestrial environment [53].

The first effort to place a space station into orbit began with the Skylab programme (1973-1974, Figure C.5), focusing on solar observations and long-duration human missions.
Despite the "space race", the U.S. and the Soviet Union initiated cooperation in 1975 with the Apollo-Soyuz Test Project. NASA's Space Shuttle programme (1981-2011, Figure C.4b) completed 135 missions with five orbiters: Columbia, Challenger, Discovery, Atlantis, and Endeavour, carrying a total of 355 individuals to space. As the first reusable spacecraft, the Shuttle enabled routine access to Low Earth Orbit (LEO), deployed and repaired satellites, supported scientific research, and contributed substantially to the construction of the ISS [53].

(a) **(b)**

Figure C.3: On the left, Mercury-Atlas 8 [54] and, on the right, Gemini-7 [55] spacecrafts.

(a)

(b)

Figure C.4: On the left, Apollo 11 Lunar Module *Eagle* on the Moon [56] and, on the right, Space Shuttle *Discovery* [57].

Figure C.5: Skylab 4 spacecraft [58].

C.1.3 Chinese programme

In 1992, the Chinese government initiated its human spaceflight programme, formulating a three-step development strategy.
The first phase aimed to launch crewed spacecraft in order to establish human spaceflight capability and conduct space-based applications and experiments.
The second phase focused on achieving key technological milestones in EVA, rendezvous and docking, and supporting short-duration human presence in orbit for applied research and operational validation.
The third and final phase set the objective of maintaining long-term operations of a permanently crewed space station, enabling a sustained human presence in space and more advanced scientific and technological activities.[59]
The first major Chinese achievement occurred in 1999 with Shenzhou-1, the nation's first uncrewed spacecraft. The first crewed mission followed in 2003 with Shenzhou-5, marking China's emergence as the third nation in history to independently place humans into space. In 2011, Tiangong-1 was launched to demonstrate rendezvous, docking, and short-term crewed habitation capabilities, followed by Tiangong-2 in 2016. In 2021, China launched the Tianhe core module of its Tiangong modular space station, marking the beginning of the third phase of its human spaceflight programme. Additional modules were subsequently integrated, and the station achieved full operational capability in 2022.
The Tiangong space station, as shown in Figure C.6, now supports a continuous human presence LEO, hosting rotating crews and enabling a broad range of scientific and technological experiments [35].

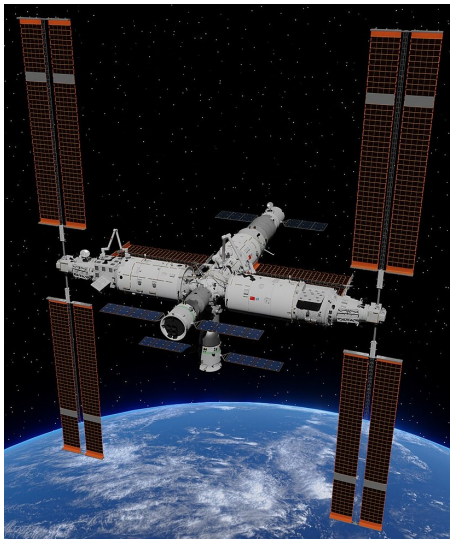


Figure C.6: Tiangong Space Station [60].

C.2 The International Space Station

The International Space Station (ISS) is a crewed orbital outpost in LEO, designed primarily for scientific research and technological development. It represents a joint effort by five major space agencies: NASA (United States), Roscosmos (Russia), ESA (Europe), JAXA (Japan), and CSA-ASC (Canada).
The ISS emerged from the convergence of multiple space station initiatives conceived during the final decades of the Cold War.
In the early 1980s, NASA launched the Space Station Freedom project as an American response to the Soviet Salyut and Mir programmes. However, economic constraints in the post-Cold-War period, coupled with the dissolution of the Soviet Union, encouraged a shift from competition to cooperation, with the objective of constructing a shared orbital station.
By the early 1990s, the United States had formalised collaboration with Russia, Europe, Japan, and Canada on a common space station referred to as "Alpha". The agreement leveraged existing technological developments from each partner. Consequently, the final ISS design incorporated key elements of NASA's Freedom programme, Russia's Mir-2 station, ESA's Columbus laboratory, and Japan's Kibo laboratory.
The construction of the ISS began in 1998, and it has since evolved into the largest and most complex human-made structure in orbit, operating as the cornerstone of sustained human presence in space and multinational space cooperation [61].

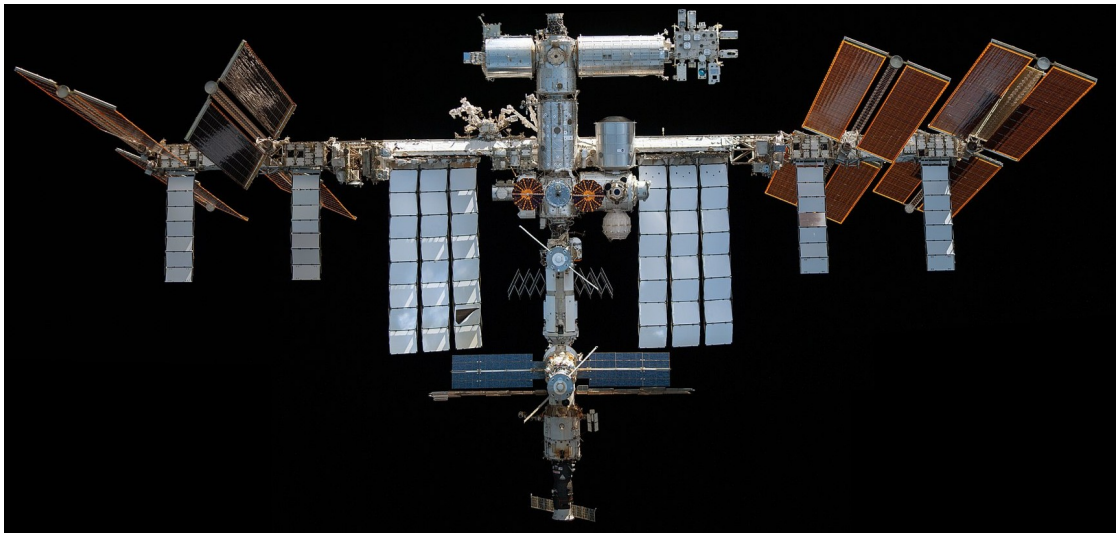
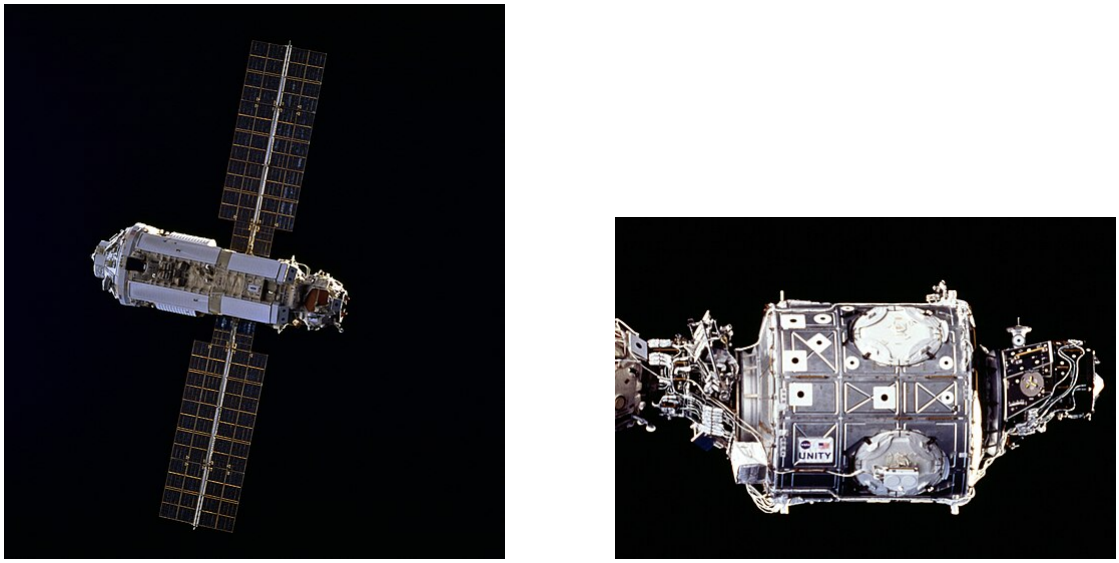


Figure C.7: International Space Station [62].

C.2.1 Development history

The assembly of the ISS began in November 1998 and proceeded as a highly complex, staged engineering effort involving multiple launch systems and international partners. Modules belonging to the Russian Orbital Segment were primarily launched atop Proton rockets and docked autonomously, with the exception of *Rassvet*, while the Space Shuttle fleet delivered most of the United States and international elements, with installation conducted via remote robotic operations or EVAs.

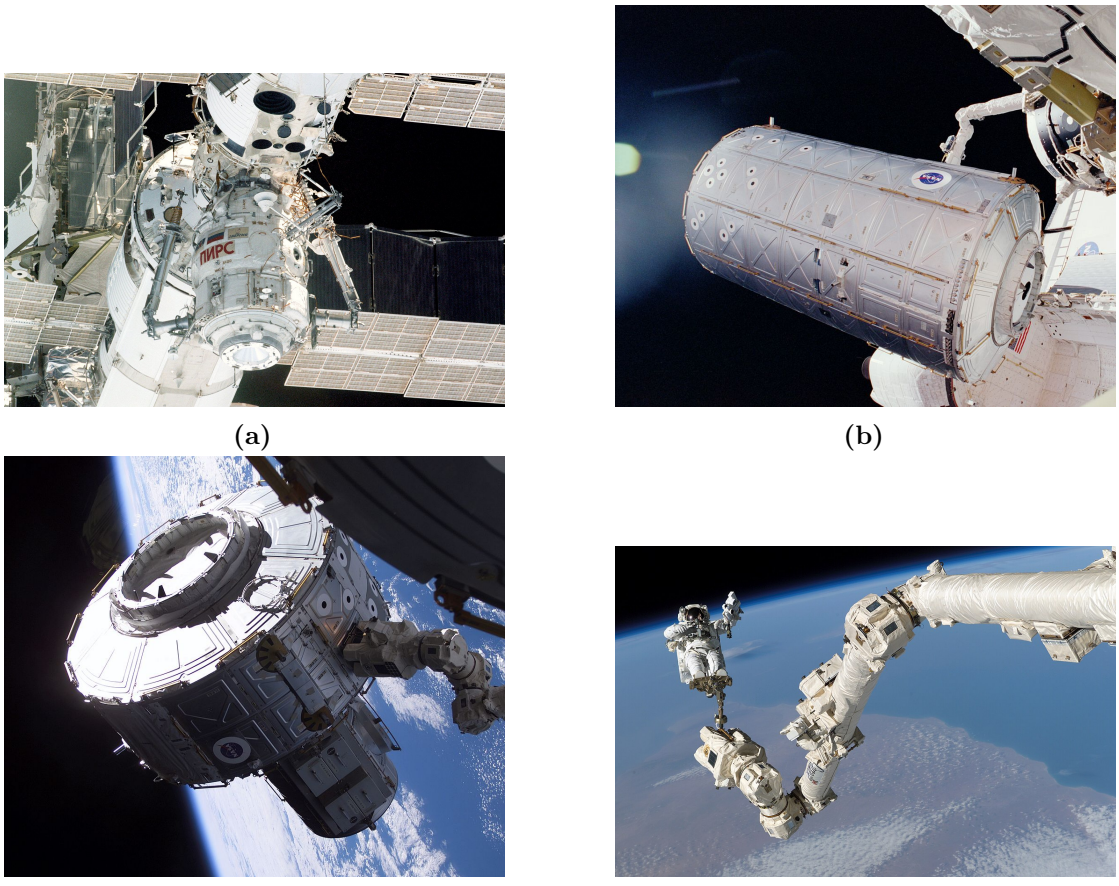
The foundation of the ISS was established with the Russian *Zarya* module (Figure C.8a), which provided power, attitude control, and communications. In December 1998, delivered by Space Shuttle Endeavour, the American *Unity* node (Figure C.8b) was attached to Zarya. It acted as the structural and functional interface between the U.S. and Russian segments and contained docking ports for future expansion. At that stage, the station lacked life-support capabilities and therefore remained uncrewed while the Russian Mir station continued regular operations.



(a) **(b)**

Figure C.8: On the left, Zarya module [63] and, on the right, Unity node [64].

A turning point occurred in July 2000 with the launch of *Zvezda*, which introduced living quarters, environmental control systems, and long-duration support capability. Continuous human presence began with Expedition 1 in November 2000, meanwhile, Space Shuttle missions began assembling the Integrated Truss Structure, adding power generation, attitude control, and high-bandwidth communications systems critical for future development.
Between 2001 and 2003, further key components arrived: the Russian *Pirs module*, the U.S. *Destiny laboratory*, the *Quest airlock*, and the *Canadarm2* robotic manipulator, alongside additional truss segments.

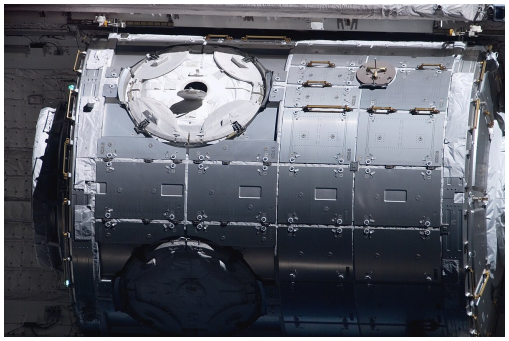


(a) **(b)**

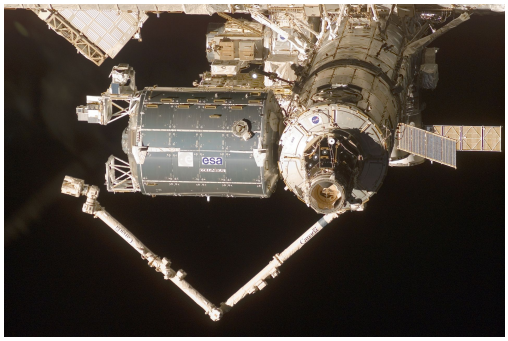
(c) **(d)**

Figure C.9: On the top left, Pirs module [65], on the top right, Destiny laboratory [66]; on the bottom left, Quest airlock [67], and, on the bottom right, Canadarm2 [68].

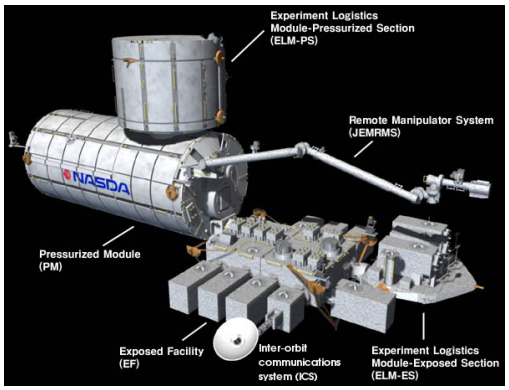
The Columbia disaster in 2003 halted Shuttle flights for nearly three years, temporarily slowing construction but not halting ISS operations. Assembly resumed in 2006, allowing installation of additional solar arrays and truss elements, followed by the *Harmony node*, the European *Columbus laboratory*, and components of Japan's *Kibō laboratory*, shortly followed by the Russian *Poisk module*.



(a)



(b)

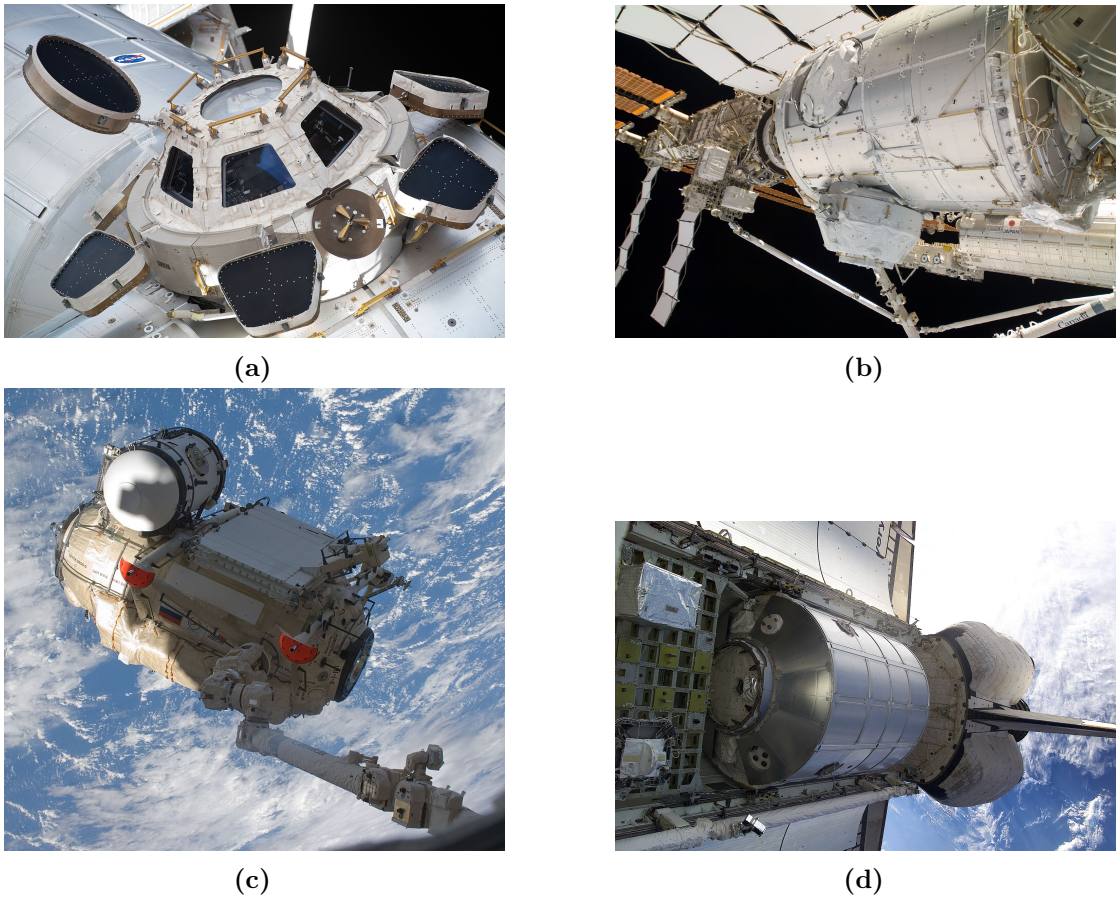


(c)

(d)

Figure C.10: On the top left, Harmony node [69], on the top right, Columbus laboratory [70]; on the bottom left, Kibō laboratory [71], and, on the bottom right, Poisk module [72].

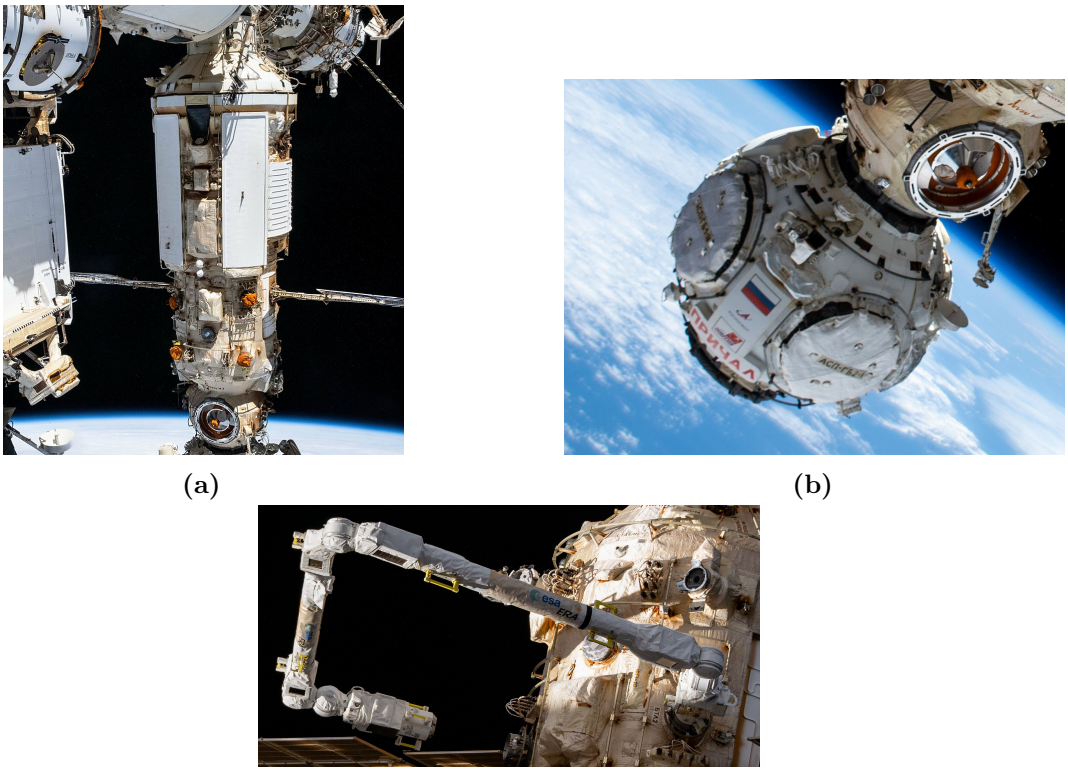
The truss structure reached completion in March 2009, enabling full power capability and major scientific expansion.
Subsequent additions included the *Cupola module*, the *Tranquility node*, the *Rassvet module*, and the *Leonardo Permanent Multipurpose Module*.



(a) **(b)** **(c)** **(d)**

Figure C.11: On the top left, Cupola module [73], on the top right, Tranquility node [74]; on the bottom left, Rassvet module [75], and, on the bottom right, Leonardo Permanent Multipurpose Module [76].

In 2021, Russia added the *Nauka module* and the *Prichal docking node* , alongside the *European Robotic Arm* (ERA). As of June 2025, NASA reports that the ISS comprises 43 pressurised and unpressurised elements [61, 77].



(a) **(b)**

(c)

Figure C.12: On the top left, Nauka module [78], on the top right, Prichal module [79] and, on the bottom, European robotic arm (ERA) [80].

Bibliography

[1] Miriam Calò. «Model Based Systems Engineering Applied to Small Satellite Systems». Politecnico di Torino, 2020 (cit. on pp. 4, 8, 10).

[2] Marshall Pratt and Matthew Dabkowski. «Analyzing the Integration of MBSE Approaches within the Aerospace Industry According to UTAUT». In: *Industrial and Systems Engineering Review* 10.2 (2022), pp. 127–134. DOI: 10.37266/ISER.2022v10i2.pp127-134 (cit. on pp. 4, 10).

[3] Azad M. Madni and Shatad Purohit. «Economic Analysis of Model-Based Systems Engineering». In: *Systems* 7.1 (2019), p. 12. DOI: 10.3390/systems7010012. URL: https://doi.org/10.3390/systems7010012 (cit. on p. 5).

[4] Morayo Adedjouma, Thibaud Thomas, Chokri Mraidha, Sebastien Gerard, and Guillaume Zeller. *From Document-Based to Model-Based System and Software Engineering*. 2016 (cit. on pp. 6, 8).

[5] David Kaslow, Bradley Ayres, Philip T. Cahill, Laura Hart, and Rose Yntema. «A Model-Based Systems Engineering (MBSE) Approach for Defining the Behaviors of CubeSats». In: *2017 IEEE Aerospace Conference*. Mar. 2017, pp. 1–14. DOI: 10.1109/AERO.2017.7943865 (cit. on p. 6).

[6] Iqtiar Md Siddique. «International Journal of Geoinformatics Science and Technology». In: *International Journal of Geoinformatics Science and Technology* (2025). DOI: 10.46610/ijgst (cit. on pp. 6, 7).

[7] Paul Logan, David Harvey, and Daniel Spencer. «Documents Are an Essential Part of Model Based Systems Engineering». In: *INCOSE International Symposium* 22.1 (2012), pp. 1899–1913. DOI: 10.1002/j.2334-5837.2012.tb01445.x (cit. on p. 8).

[8] Michele Cencetti. «Evolution of Model-Based System Engineering Methodologies for the Design of Space Systems in the Advanced Stages of the Project (Phases B-C)». Politecnico di Torino, 2013 (cit. on p. 9).

[9] Kaitlin Henderson and Alejandro Salado. «Value and Benefits of Model-Based Systems Engineering (MBSE): Evidence from the Literature». In: *Systems Engineering* 24.1 (2021), pp. 51–66. DOI: 10.1002/sys.21566 (cit. on p. 9).

[10] Matthias Raif, Ulrich Walter, and Jasper Bouwmeester. «Dynamic system simulation of small satellite projects». In: *Acta Astronautica* 67.9-10 (2010), pp. 1138–1156. DOI: 10.1016/j.actaastro.2010.06.038 (cit. on p. 10).

[11] Alberto Gonzalez Fernandez, Elaheh Maleki, Nils Fischer, Evelyn Honoré-Livermore, Nikolena Christofi, and Marcel Verhoef. «The European Space Agency MBSE Methodology». In: *INCOSE International Symposium* 34.1 (2024), pp. 2069–2086. DOI: 10.1002/iis2.13256 (cit. on p. 11).

[12] Jon B. Holladay, Jessica Knizhnik, Karen J. Weiland, Amanda Stein, Terry Sanders, and Paul Schwindt. «MBSE Infusion and Modernization Initiative (MIAMI): "Hot" Benefits for Real NASA Applications». In: *2019 IEEE Aerospace Conference*. Mar. 2019, pp. 1–14. DOI: 10.1109/AERO.2019.8741795 (cit. on p. 12).

[13] Iqtiar Md Siddique. «MBSE Implementation in Small Satellite Systems: Rationale for Adoption over Traditional Document-Based Systems Engineering». In: *International Journal of Geoinformatics Science and Technology* 1.1 (2025), pp. 1–13. DOI: 10.46610/IJGST.2025.v01i01.001 (cit. on p. 13).

[14] David Kaslow, Grant Soremekun, Hongman Kim, and Sara Spangelo. «Integrated model-based systems engineering (MBSE) applied to the Simulation of a CubeSat mission». In: *2014 IEEE Aerospace Conference*. 2014, pp. 1–14. DOI: 10.1109/AERO.2014.6836317 (cit. on p. 13).

[15] Carlotta Deiana. «Integrating Data-Driven and Model-Based Systems Engineering for End-to-End Small-Satellite Design». Master's thesis. Torino, Italy: Politecnico di Torino, 2025. URL: http://webthesis.biblio.polito.it/id/eprint/36821 (cit. on p. 13).

[16] Awele I Anyanhun and William W Edmonson. «Inter-satellite communication MBSE design framework for small satellites». In: *2017 Annual IEEE International Systems Conference (SysCon)*. 2017, pp. 1–7. DOI: 10.1109/SYSCON.2017.7934707 (cit. on p. 14).

[17] Matthew Hause. «The SysML Modelling Language». In: *Fifth European Systems Engineering Conference*. Sept. 2006. URL: https://www.researchgate.net/publication/260359652_The_OMG_Modelling_Language_SYSML (cit. on p. 15).

[18] Pascal Roques. *Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella*. ISTE Press Ltd, 2018 (cit. on pp. 16, 17).

[19] Olivier Casse. *SysML in Action with Cameo Systems Modeler. Implementation of Model Based System Engineering Set*. ISTE Press Ltd, 2017 (cit. on pp. 18, 19).

[20] Christopher B. Watkins, Jerry Varghese, Michael Knight, Becky Petteys, and Jordan Ross. «System Architecture Modeling for Electronic Systems Using MathWorks System Composer and Simulink». In: *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. Oct. 2020, pp. 1–10. DOI: 10.1109/DASC50938.2020.9256753 (cit. on pp. 19, 20, 24).

[21] Thierry Le Sergent, Alain Le Guennec, François Terrier, Yann Tanguy, and Sébastien Gérard. «SCADE System, a Comprehensive Toolset for Smooth Transition from Model-Based System Engineering to Certified Embedded Control and Display Software». In: *ERTS 2012 Proceedings*. Toulouse, France, Feb. 2012. URL: https://hal.science/hal-02192066 (cit. on p. 21).

[22] Ansys. *Diamond diagram*. Accessed: 2025-09-02. 2015. URL: https://cdn.leapaust.com.au/wp-content/uploads/2023/04/06180448/SCADE-Suite-Diamond-2015-Small.png.webp (cit. on p. 21).

[23] Maxime Perrotin, Eric Conquet, Julien Delange, André Schiele, and Thanassis Tsiodras. «TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future». In: *SDL 2011: Integrating System and Software Modeling*. Ed. by Iulian Ober and Ileana Ober. Vol. 7083. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-25264-8_4 (cit. on p. 22).

[24] NASA. *NASA Systems Engineering Handbook*. Tech. rep. NASA/SP-2007-6105 Rev1. NASA Headquarters, 2008. URL: https://nasa.gov/sites/default/files/atoms/files/nasa_systems_engineering_handbook.pdf (cit. on p. 32).

[25] MathWorks. *Define Sequence Diagrams*. Accessed: 2025-09-10. 2025. URL: https://www.mathworks.com/help/systemcomposer/ug/define-sequence-diagrams.html (cit. on p. 32).

[26] MathWorks. *Author Sequence Diagrams*. Accessed: 2025-09-10. 2025. URL: https://www.mathworks.com/help/systemcomposer/author-sequence-diagrams.html (cit. on p. 32).

[27] MathWorks. *Sequence Diagram Fragments for Message Ordering*. Accessed: 2025-09-10. 2025. URL: https://www.mathworks.com/help/systemcomposer/ug/sequence-diagram-fragments-for-message-ordering.html (cit. on p. 32).

[28] MathWorks. *Use Sequence Diagrams in the Views Editor*. Accessed: 2025-09-10. 2025. URL: https://www.mathworks.com/help/systemcomposer/ug/use-sequence-diagrams-in-the-views-editor.html (cit. on p. 32).

[29] Carlton Douglas et al. *Requirements Definition For Space Science Missions*. Accessed: 2025-09-11. 2016. URL: https://uasal.github.io/spacehardwarehandbook-public/requirements_management.html (cit. on p. 35).

[30] MathWorks. *Link, Manage, and Verify Requirements*. Accessed: 2025-09-10. 2025. URL: https://www.mathworks.com/help/systemcomposer/link-and-manage-requirements.html (cit. on pp. 35, 37, 39, 40).

[31] NASA. *4.1 Stakeholder Expectations Definition - NASA*. Accessed: 2025-09-05. 2023. URL: https://www.nasa.gov/reference/4-1-stakeholder-expectations-definition/ (cit. on p. 44).

[32] NASA Jet Propulsion Laboratory. *The Science Traceability Matrix*. https://science.nasa.gov/wp-content/uploads/2023/04/Launchpad_Session3_STM_18Nov2019_smf_final.pdf. Accessed: 2025-10-02. Nov. 2019 (cit. on p. 49).

[33] NASA. *NASA Systems Engineering Handbook*. Rev 2. NASA/SP-2016-6105 Rev2. NASA/SP-2016-6105 Rev2. National Aeronautics and Space Administration. Washington, DC, 2016. URL: https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook (cit. on p. 51).

[34] Ronald D. Reed and Gary R. Coulter. «Physiology of Spaceflight». In: *Human Spaceflight: Mission Analysis and Design*. Ed. by Wiley J. Larson and Linda K. Pranke. New York: McGraw-Hill, 1999. Chap. 5, pp. 103–130. ISBN: 0-07-236811-X (cit. on pp. 73–75, 77–79, 81).

[35] Loris Giannini. «Integrated Model-Based Systems Engineering methodology for design, analysis and simulation of an Environmental Control and Life Support System for an Analog Habitat». Corso di laurea magistrale in Ingegneria Aerospaziale, classe LM-20. Tesi di laurea magistrale. Torino, Italia: Politecnico di Torino, 2025. URL: https://webthesis.biblio.polito.it/36827/ (cit. on pp. 73–76, 83–85, 90, 103, 107, 139).

[36] P. M. Jones and E. Fiedler. *Human Factors in Space Exploration*. Technical Report NASA/TM-2011-11268. Chapter draft for *Review of Human Factors and Ergonomics*, Volume 6. NASA Ames Research Center, 2011 (cit. on p. 73).

[37] P. O. Wieland. *Designing for Human Presence in Space: An Introduction to Environmental Control and Life Support Systems*. Vol. 1324. NASA Reference Publication 1324. Washington, D.C.: National Aeronautics, Space Administration, Office of Management, Scientific, and Technical Information Program, 1994 (cit. on pp. 74–76, 79, 88, 98, 103, 104, 107, 111, 115, 119).

[38] S. Scarsoglio. *Physiology Alterations Associated to Human Space Flight*. Course slides for "Biofluid Dynamics and Space Medicine", Politecnico di Torino. 2024 (cit. on pp. 80–82).

[39] S. Corpino. *Space Environment and Its Effects on Humans*. Course slides for "Aerospace Systems", Politecnico di Torino. 2024 (cit. on p. 80).

[40] European Cooperation for Space Standardization (ECSS). *ECSS-E-ST-34C: Environmental Control and Life Support (ECLS)*. Tech. rep. ECSS-E-ST-34C. Issued 31 July 2008, Revision C. European Cooperation for Space Standardization, July 2008. URL: https://ecss.nl/wp-content/uploads/standards/ecss-e/ECSS-E-ST-34C31July2008.pdf (cit. on p. 83).

[41] P. Eckart. *Spaceflight Life Support and Biospherics*. Space Technology Library 5. Torrance, California: Microcosm Press, 1996 (cit. on pp. 83–85, 94).

[42] NASA. *Space Station Regenerative ECLSS Flow Diagram*. Accessed: 2025-11-14; Public domain (U.S. Government work). 2008. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/SpaceStationCycle.svg/1920px-SpaceStationCycle.svg.png (cit. on p. 84).

[43] National Aeronautics and Space Administration. *International Space Station Familiarization*. Tech. rep. Lyndon B. Johnson Space Center, Houston, TX, USA: Mission Operations Directorate, Space Flight Training Division, July 1998 (cit. on pp. 86–89, 92).

[44] Bethany M. Clement. *Crew Health Care System (CHeCS) Design Research, Documentation, and Evaluations*. Tech. rep. JSC-CN-23601. NASA Johnson Space Center, 2011. URL: https://ntrs.nasa.gov/api/citations/20110011351/downloads/20110011351.pdf (cit. on pp. 89, 90).

[45] Charles Ray and Alan Adams. *Environmental Control and Life Support System*. Tech. rep. NASA Document ID 19930018529. Presented at Technology for Space Station Evolution Workshop, 16–19 January 1990, Dallas, Texas. NASA Marshall Space Flight Center, Jan. 1990. URL: https://ntrs.nasa.gov/api/citations/19930018529/downloads/19930018529.pdf (cit. on p. 90).

[46] Paul O. Wieland. *Living Together in Space: The Design and Operation of the Life Support Systems on the International Space Station*. Tech. rep. NASA/TM–1998-206956. Huntsville, Alabama: NASA Marshall Space Flight Center, 1998. URL: https://ntrs.nasa.gov/api/citations/19990023237/downloads/19990023237.pdf (cit. on pp. 90, 92, 103).

[47] Susan Doll and Peter Eckart. «Physiology of Spaceflight». In: *Human Spaceflight: Mission Analysis and Design*. Ed. by Wiley J. Larson and Linda K. Pranke. New York: McGraw-Hill, 1999. Chap. 17, pp. 539–573. ISBN: 0-07-236811-X (cit. on pp. 94, 98).

[48] Gilles Clément and Angelia P. Bukley. «Human space exploration – From surviving to performing». In: *Acta Astronautica* 100 (July 2014), pp. 101–106. ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2014.04.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0094576514001180 (cit. on p. 135).

[49] Laurence Tognetti. *A brief history of Soviet and Russian human spaceflight*. Accessed: 2025-10-29. Astronomy Magazine. 2023. URL: https://www.astronomy.com/space-exploration/a-brief-history-of-soviet-and-russian-human-spaceflight/ (cit. on p. 136).

[50] Pline. *Mock-up of the spacecraft Vostok-1 (1961), Musée de l'Air et de l'Espace, Paris Le Bourget*. Retrieved from Wikimedia Commons, licensed under CC BY-SA 2.5/3.0 and older GFDL licences. 2009. URL: https://commons.wikimedia.org/wiki/File:Vostok-1-musee-du-Bourget-P.jpg (visited on 10/27/2025) (cit. on p. 136).

[51] NASA Astronaut in Duty. *Soyuz MS spacecraft departs the International Space Station*. Immagine modificata digitalmente, rilasciata sotto licenza CC0 1.0 Universal (dominio pubblico). 2022. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/f/fa/Soyuz_MS.jpg/500px-Soyuz_MS.jpg (visited on 10/27/2025) (cit. on p. 136).

[52] NASA. *Mir space station viewed from Space Shuttle Atlantis during mission STS-81*. Accessed: 2025-10-29. Wikimedia Commons. 1997. URL: https://upload.wikimedia.org/wikipedia/commons/e/ef/Mir_from_STS-81.jpg (cit. on p. 136).

[53] NASA. *60 Years and Counting – Human Spaceflight*. Accessed: 2025-10-29. NASA. 2021. URL: https://www.nasa.gov/specials/60counting/spaceflight.html (cit. on p. 137).

[54] NASA. *Mercury 8 in Hangar*. Immagine di pubblico dominio, NASA GPN-2000-001441. 1962. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/3/3d/Mercury_8_in_Hangar_-_GPN-2000-001441.jpg/800px-Mercury_8_in_Hangar_-_GPN-2000-001441.jpg (visited on 10/27/2025) (cit. on p. 137).

[55] Mark Avino. *Gemini VII Capsule in Restoration Hangar*. Immagine di pubblico dominio, NASA GPN-2000-001441. 2019. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/e/ef/NASM-A19680273000-NASM2019-10008.jpg/800px-NASM-A19680273000-NASM2019-10008.jpg (visited on 10/27/2025) (cit. on p. 137).

[56] Neil Armstrong. *Buzz Aldrin and the Apollo 11 Lunar Module, AS11-40-5927*. Accessed: 2025-11-03. NASA / Wikimedia Commons. 1969. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/Buzz_Aldrin_and_Apollo_11_Lunar_Lander%2C_AS11-40-5927.jpg/800px-Buzz_Aldrin_and_Apollo_11_Lunar_Lander%2C_AS11-40-5927.jpg (cit. on p. 138).

[57] NASA. *Space Shuttle Discovery on mission STS-121*. Accessed: 2025-11-03. NASA / Wikimedia Commons. 2006. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/4/4c/STS-121-DiscoveryEnhanced.jpg/1920px-STS-121-DiscoveryEnhanced.jpg (cit. on p. 138).

[58] NASA. *Skylab and Earth Limb*. 2000. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/9/92/Skylab_and_Earth_Limb_-_GPN-2000-001055.jpg/1024px-Skylab_and_Earth_Limb_-_GPN-2000-001055.jpg (visited on 10/28/2025) (cit. on p. 138).

[59] China Manned Space Agency. *About CMS*. Accessed: 2025-11-03. China Manned Space Agency. URL: https://en.cmse.gov.cn/aboutcms/ (cit. on p. 139).

[60] China Manned Space Agency (CMSA). *Chinese Tiangong Space Station*. Image of the Tiangong space station in orbit. (Visited on 10/28/2025) (cit. on p. 139).

[61] Wikimedia Foundation. *International Space Station*. Accessed: 2025-11-04. URL: https://en.wikipedia.org/wiki/International_Space_Station (cit. on pp. 140, 145).

[62] NASA. *The station pictured from the SpaceX Crew Dragon*. Public domain (U.S. Government work). Photo ID: JSC2021-E-064215; fly-around after undocking from Harmony module's space-facing port, 8 Nov 2021. (Visited on 10/28/2025) (cit. on p. 140).

[63] NASA. *Zarya module as seen during STS-88*. Accessed: 2025-11-04; Public domain, US government work. Wikimedia Commons / NASA. 1998. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/d/db/Zarya_from_STS-88.jpg/500px-Zarya_from_STS-88.jpg (cit. on p. 141).

[64] NASA. *Unity Module of the International Space Station*. Accessed: 2025-11-04; Public domain, US government work. Wikimedia Commons / NASA. 1998. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/a/ac/ISS_Unity_module.jpg/500px-ISS_Unity_module.jpg (cit. on p. 141).

[65] NASA. *STS-110 "Sts110-363-001" image*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e5/Sts110-363-001.jpg/1920px-Sts110-363-001.jpg (cit. on p. 142).

[66] NASA. *ISS Destiny Laboratory Module*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/7/76/ISS_Destiny_Lab.jpg/1125px-ISS_Destiny_Lab.jpg (cit. on p. 142).

[67] NASA. *ISS Quest Airlock*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/ISS_Quest_airlock.jpg/1071px-ISS_Quest_airlock.jpg (cit. on p. 142).

[68] NASA. *Astronaut Stephen K. Robinson anchored to a foot-restraint on the Canadarm2 of the International Space Station during EVA3, STS-114*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e5/STS-114_Steve_Robinson_on_Canadarm2.jpg/1200px-STS-114_Steve_Robinson_on_Canadarm2.jpg (cit. on p. 142).

[69] NASA. *Harmony Node (Node 2) in the payload bay of Space Shuttle Discovery during STS-120*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/STS-120_Harmony_in_Discovery%27s_payload_bay.jpg/1200px-STS-120_Harmony_in_Discovery%27s_payload_bay.jpg (cit. on p. 143).

[70] NASA. *STS-122 docked Columbus*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/STS-122_docked_Columbus.jpg/1920px-STS-122_docked_Columbus.jpg (cit. on p. 143).

[71] NASA/JAXA. *Japanese Experiment Module "Kibo" on the International Space Station*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/0/07/Japanese_Experiment_Module_Kibo.jpg (cit. on p. 143).

[72] NASA. *Poisk (MRM-2) module on the International Space Station*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/a/a3/Poisk.Jpeg (cit. on p. 143).

[73] NASA Johnson Space Center. *The International Space Station's "window to the world" (Cupola) from Nauka*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/2/22/Iss065e241659.jpg/1920px-Iss065e241659.jpg (cit. on p. 144).

[74] NASA. *PMA-3 relocation during STS-130*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/8/8e/STS-130_PMA-3_relocation_3.jpg/1920px-STS-130_PMA-3_relocation_3.jpg (cit. on p. 144).

[75] NASA. *Rassvet module (MRM-1) and Canadarm2 on the International Space Station (cropped view)*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/7/7d/Rassvet_Canadarm_Crop.jpg/975px-Rassvet_Canadarm_Crop.jpg (cit. on p. 144).

[76] NASA. *Leonardo Multi-Purpose Logistics Module (MPLM) in the payload bay of Space Shuttle Discovery*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/0/01/Mplm_in_shuttle.jpg/1920px-Mplm_in_shuttle.jpg (cit. on p. 144).

[77] NASA. *International Space Station Assembly Elements*. Accessed: 2025-11-04. NASA. 2010. URL: https://www.nasa.gov/international-space-station/international-space-station-assembly-elements/ (cit. on p. 145).

[78] NASA. *Nauka Module as seen from Cupola during VKD-51 spacewalk*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Nauka_Module_as_seen_from_Cupola_during_VKD-51_spacewalk.jpg/800px-Nauka_Module_as_seen_from_Cupola_during_VKD-51_spacewalk.jpg (cit. on p. 145).

[79] NASA. *Prichal seen during Expedition 66 spacewalk (ISS066-E-119955)*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/b/b5/Prichal_seen_during_Expedition_66_spacewalk_%28ISS066-E-119955%29.jpg/1280px-Prichal_seen_during_Expedition_66_spacewalk_%28ISS066-E-119955%29.jpg (cit. on p. 145).

[80] NASA/ESA. *The European Robotic Arm extends out from the Nauka module (ISS067E034865) (cropped)*. Accessed: 2025-11-06. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/3/39/The_European_robotic_arm_extends_out_from_the_Nauka_module_%28iss067e034865%29_%28cropped%29.jpg/1920px-The_European_robotic_arm_extends_out_from_the_Nauka_module_%28iss067e034865%29_%28cropped%29.jpg (cit. on p. 145).

Acknowledgements

At the end of this long, rewarding and fulfilling thesis journey, I would like to take a moment to thank those who made this work possible.

First and foremost, I would like to thank Professor Corpino and Dr Campioli who, over these past months, not only made this dissertation possible but also guided me through every stage of the process with clear and wise suggestions.

I would also like to express my deepest gratitude to my parents, the first supporters of this achievement and direct witnesses to years of studies. I'll be forever grateful that you believe in me.

Last but not least, I would like to thank all the friends whose paths have crossed mine, even if only briefly. In one way or another, each one of you has contributed to this milestone.