POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Master's Degree Thesis

Satellite Digital Twin for In Orbit Servicing Mission

Supervisors

Prof. Paolo MAGGIORE

Tutor Carlo Maria PACCAGNINI

Candidate

Filippo GOFFREDO

07 2025

Summary

This thesis focuses on the development and testing of an Electrical Power System (EPS) C++ model for the In-Orbit Servicing (IOS) mission. It begins with an introduction outlining the objectives of the IOS mission, followed by a general overview of the Generic Application Security Framework and the security protocols used for telemetry and telecommand operations in space missions. The state-of-the-art chapter presents the technologies that were considered during the development phase, including both those adopted and those ultimately not used. Afterward, Chapter 3 describes the development environment, while the following sections detail the design and implementation phases. This part of the work focuses in particular on the battery model, the Power Line Interface, and the MILBUS1553 interface, also explaining the physical characteristics and parameters that were used to extract and validate the corresponding models. Finally, Chapter 5 discusses the testing procedures and compares the obtained results with real-world data and the expected performance defined in the documentation provided by Thales Alenia Space. The final chapter summarizes the conclusions and future work for possible improvements.

Table of Contents

Chapter 1

Introduction

1.1 Context - What is a Digital Twin and why it's foundamental for security

This thesis presents the development of a new satellite simulator for In-Orbit Servicing (IOS) missions, an Italian Space Agency (ASI) project led by Thales Alenia Space. This work investigates the integration of advanced technologies and methodologies to enhance satellite operations and mission planning. The introduction provides a comprehensive overview of the research topic, its primary objectives, and the significance of the study, thereby establishing a foundation for the subsequent chapters.

They serve as an indispensable tool for testing, validation, and training. A simulation environment functions not only as a platform for pre-launch verification but also as a crucial asset for validating new versions of On-Board Software (OBSWW) before their deployment on a live satellite. This rigorous process is essential for minimizing the risk of mission-critical failures, a fundamental concern in the field of aerospace engineering. The simulator's fidelity, therefore, must closely approximate that of the real satellite, ensuring that simulation results provide a reliable basis for predicting the behavior and performance of the actual spacecraft. Consequently, the data acquired during a simulation is considered a trustworthy source for informed engineering

and operational decisions. The implementation of this high fidelity is a significant technical challenge, requiring meticulous attention to detail in modeling physical phenomena, from orbital dynamics to the subtle electrical properties of components.

The application of simulation technology extends beyond pre-launch activities. It is particularly vital for missions such as In-Orbit Servicing, where OBSW updates may be required to adapt to unforeseen circumstances or to prolong the operational lifespan of the spacecraft. A high-fidelity simulator enables comprehensive regression testing of new software, meticulously verifying that enhancements do not inadvertently introduce faults or compromise performance. The ability to perform such extensive testing in a controlled environment is invaluable, as it prevents costly and potentially mission-ending issues that could arise from deploying untested software updates in orbit.

Throughout the satellite production lifecycle, the simulator performs multiple critical functions. It is employed during the early design phases to test the functionality of satellite subsystems and is subsequently used to validate the OBSW before its deployment. Furthermore, the simulator is an essential tool for training operators and engineers, providing a secure environment for practicing complex and time-sensitive procedures, ranging from routine maneuvers to responding to critical emergency scenarios. The realism of the training environment, enabled by the simulator's high fidelity, is crucial for building the operational competence and confidence of ground control teams.

This research focuses predominantly on the development of the Electrical Power System (EPS) simulator, a subsystem of utmost importance to the satellite. The EPS simulator is responsible for modeling the satellite's power generation, storage, and distribution, thereby guaranteeing a consistent and reliable power supply throughout the mission. A stable and accurate power model is a prerequisite for the meaningful simulation of other subsystems, such as propulsion or attitude control, whose operational capabilities are intrinsically linked to power availability. Therefore, a robust EPS simulation represents a foundational element for the entire virtual spacecraft model, providing the energy backbone that dictates the functionality and performance of all other

simulated systems.

1.2 Cybersecurity in Space

It is essential not only to create an environment that has an high fidelity to real life architecture but also it is foundamental to make sure that the simulator and the models follow a standardized path of development so that the models can be considered scalable. It is also very critical the security behind the development of the simulator, the coding language used and the architecture of the simulator itself. Cybersecurity might seem like a secondary concern in this context but it isn't because if the simulator is compromised, the entire mission could be at risk. A security breach could allow unauthorized manipulation of the simulator's models or data, leading to the generation of incorrect telemetry, corrupted telecommands, or misleading system responses. Such issues could propagate into the operational phase by influencing mission planning, validation processes, or even onboard software, potentially resulting in mission failure or damage to spacecraft hardware. For this reason, during the development of the simulator, several essential coding practices were applied to ensure that the code remains secure and that the simulator's architecture is robust against cyberattacks and data tampering.

In 2013, ESA started and Agency-internal activity on Secure Software Engineering (SSE) with the participation of several ESA directorates and projects. The main objective of this activity has been to standardise secure software engineering processes on top of existing European Cooperation for Space Standardisation (ECSS) software engineering and product assurance standards and to provide practical guidance to the ESA software engineering practitioners supporting effective and efficient implementation of these SSE practices and processes. The standard has been developed first internally for ESA but the plan is to evolve it to ECSS level. The main outputs of this activity are several documents: [1]

- A Secure Software Engineering Gap Analysis Technical Note that

documents gaps found between the ECSS standards and secure software and systems engineering best practices;

- An Internal Secure Software Engineering Standard that specifies and formalises secure software engineering processes on the basis of the ECSS E-40 and Q-80 software engineering standards;
- An Internal Secure Software Engineering Handbook of guidelines for implementation of the standard;
- A Glossary of Secure Software Engineering Terms;
- A Baseline Catalogue of Security Requirements that contains security requirements to be used during the security requirements specification process as defined in the standard.

1.2.1 Generic Application Security Framework (GASF)

ESA, together with industry, has developed the Generic Application Security Framework (GASF) to support the software requirements engineering processes defined in the SSE standard. The framework provides both a catalogue of software security requirements and tools that help technical officers and developers select those relevant to a given project. Since the tool operates independently of the catalogue, it can also be used with new catalogues from SSE standardization or even in other domains such as IT infrastructure and hardware development. The security requirements themselves vary depending on the criticality of the information handled, the formal classification of the software, its deployment environment, and any higher system-level security constraints. For every new development, the technical officer must select from a wide range of possible requirements. GASF simplifies this by using requirement templates based on security profiles. These templates act as filters: some focus on confidentiality, integrity, and availability levels determined during risk assessment; others capture the characteristics of the deployment environment such as a DMZ or internal network; and a third type incorporates project-specific

recommendations derived from system-level security engineering. All templates can be applied in combination, and the tool automatically detects conflicts or inconsistencies that the technical officer must resolve manually. Once the selection is made, the requirements can be tailored further and saved as a project template. The resulting set can then be exported in multiple formats, including Word, Excel, and CSV for import into DOORS. While further evolution of the framework is planned, the current version is already mature and, in the case of mission data systems, its use has become mandatory under ESA's Information Security Management System (ISMS). [1]

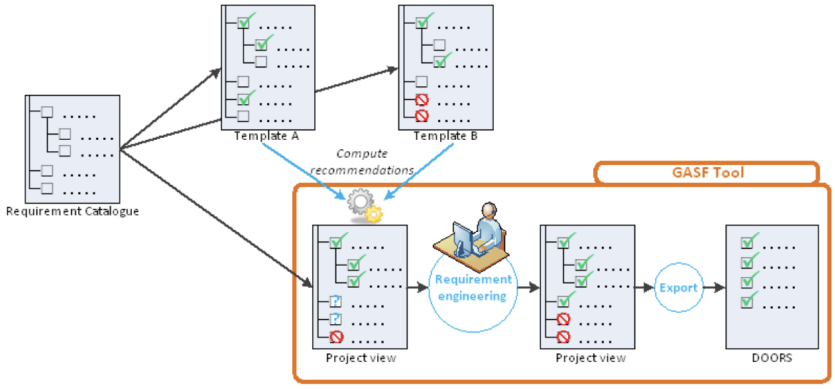

Figure 1.1: GASF Tool Requirements Specification Process. [1]

1.2.2 Security in Space transmissions for telemetries and telecommands

The SDLS Protocol is a data processing method for space missions that need to apply authentication and/or confidentiality to the contents of Transfer Frames used by Space Data Link Protocols over a space link. The Security Protocol is provided only at the Data Link Layer (Layer 2) of the OSI Basic Reference Model, as illustrated in the figure below. [2]

SDLS ensures that data exchanged between spacecraft and ground stations is protected from threats such as unauthorized access, tampering, or replay attacks. It does this by integrating cryptographic

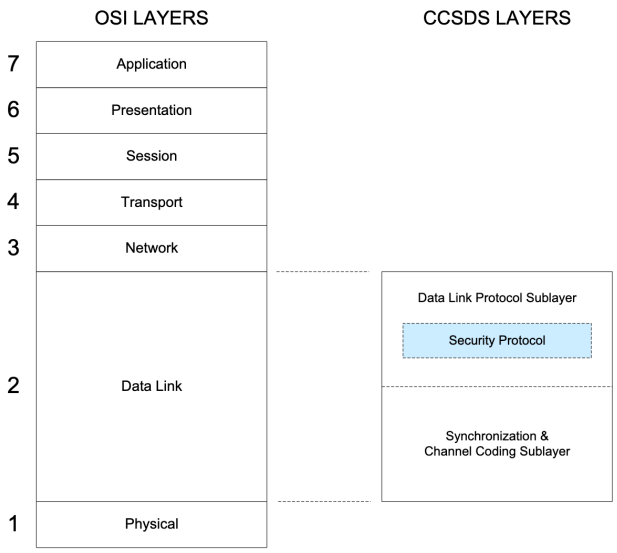

Figure 1.2: SDLS Protocol in the OSI Model. [2]

protections directly into the data link layer, independent of higher-layer applications or specific cryptographic algorithms

It can be used with several CCSDS space data link protocols:

- Telemetry (TM)
- Telecommand (TC)
- Advanced Orbiting Systems (AOS)
- Unified Space Data Link Protocol (USLP)

The protocol relies on the concept of a Security Association, which defines the cryptographic context under which communication takes place. Each Security Association is essentially a one-way session that binds together the choice of cryptographic algorithms, the secret keys, the rules for handling initialization vectors and sequence numbers, and the format of the header and trailer. To identify the correct association, every protected frame carries a Security Parameter Index in its header. This index allows the receiver to select the right cryptographic state and thus interpret the frame correctly.

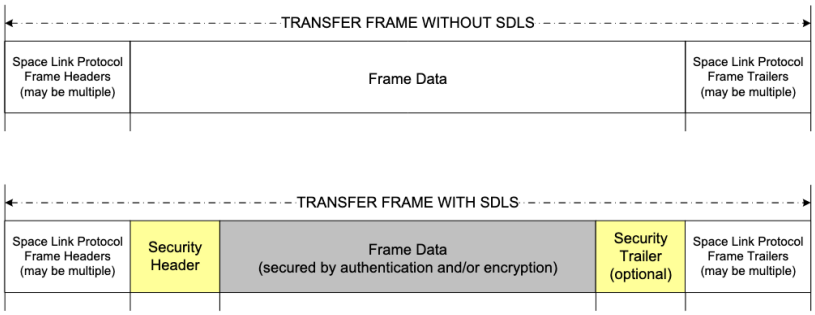

Figure 1.3: Security Protocol Interaction with Space Link Frames [2]

When a frame is prepared for transmission, the SDLS protocol intervenes through what is known as the ApplySecurity function. The partially constructed transfer frame, which already contains mission data and the standard CCSDS headers, is passed to this function. The function adds the security header, sets the sequence number, and optionally introduces an initialization vector or padding information depending on the chosen mode of operation. It then applies the selected cryptographic transformation. If the mission requires confidentiality, the mission data within the frame is encrypted. If the mission requires integrity and origin assurance, a message authentication code is computed over selected parts of the frame. If both services are required, encryption is applied first and authentication is calculated over the resulting ciphertext. The computed message authentication code, when present, is written into the trailer. The secured frame is then handed back to the data link protocol, which adds the remaining protocol-specific elements such as error control fields before transmission.

At the receiving end, the ProcessSecurity function reverses these operations. The function begins by examining the security header to extract the Security Parameter Index, which links the frame to the appropriate Security Association. Once the association has been identified, the receiver verifies that the sequence number falls within the acceptable replay window. This mechanism ensures that previously valid frames cannot be resent maliciously, since any frame with an unexpected sequence number is discarded. The receiver then checks

the message authentication code in the trailer. A valid code confirms that the frame has not been altered in transit and that it originated from an authorized sender. If encryption has been applied, the function decrypts the protected data field, restoring the original mission data for delivery to the upper protocol layers.

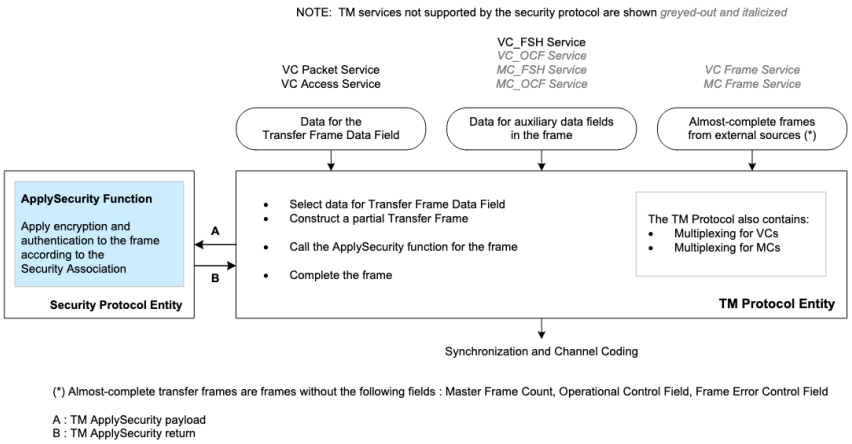

Figure 1.4: ApplySecurity and ProcessSecurity Functions [2]

The protocol supports three operational modes, namely authentication only, encryption only, and authenticated encryption. Each mode makes use of the same structural framework of headers and trailers, ensuring a consistent integration with the underlying telemetry, telecommand, AOS, or USLP transfer frames. The integrity of the frame header fields can be selectively protected by the authentication mechanism, while confidentiality is always applied exclusively to the data field.

Replay protection is a fundamental part of the protocol. Each transmitted frame carries a sequence number that is strictly monotonic within the context of a Security Association. The receiver accepts only frames whose sequence numbers fall within a defined acceptance window, which allows the system to tolerate expected transmission delays but prevents attackers from injecting old frames back into the stream.

The integration of SDLS into CCSDS protocols is designed to be minimally invasive. Synchronization and channel coding sublayers

remain unaffected, and legacy infrastructures can continue to operate with minimal adjustments. The only modifications occur within the construction and interpretation of transfer frames, which gain additional security metadata while still conforming to the original structural constraints of the CCSDS standards.

For mission planning, the lengths of the security header and trailer fields are defined as managed parameters. They remain constant throughout the mission to guarantee predictable frame sizes and to simplify hardware and software implementations. Security Associations can be preloaded before launch or activated dynamically during operations, depending on the mission profile. The management of keys, associations, and other control functions is not defined within the protocol itself but is delegated to the Extended Procedures defined in a companion standard, which allows agencies to adapt security management to their operational environment.

In practice, the Space Data Link Security protocol provides a systematic method of embedding cryptographic protection into every frame transmitted between spacecraft and ground stations. By doing so, it ensures that the data remains confidential when required, that its integrity and authenticity can always be verified, and that replayed transmissions cannot compromise the mission. This approach makes SDLS a fundamental building block for secure space communications, offering a balance between rigorous security requirements and the operational constraints of long-duration and cross-agency missions.

1.3 In Orbit Servicing Mission

In-orbit servicing (IOS) represents one of the most transformative trends in space operations and spacecraft lifecycle management. Traditionally, satellites are launched into orbit with all the propellant, hardware, and capabilities they will ever have, which imposes a strict limit on mission duration. Once the propellant is depleted or a critical subsystem fails, the satellite becomes unusable and is either left in a disposal orbit or turned into space debris. This paradigm has shaped the economics of space missions for decades, where reliability margins are pushed

to the extreme, redundancy is built into nearly every design, and replacement missions require the launch of an entirely new spacecraft costing hundreds of millions of dollars.

The concept behind IOS missions is to break this "one-shot" model of spacecraft operations by introducing servicing spacecraft that can interact physically with satellites already in orbit. These servicing spacecraft, sometimes called "space tugs," "servicers," or "robotic servicing vehicles," are equipped with advanced guidance, navigation, and control (GNC) systems, robotic manipulators, docking mechanisms, and specialized payloads. Their role is to rendezvous with client satellites, establish a controlled physical or robotic interface, and perform tasks that would traditionally require direct human intervention. By enabling servicing in orbit, IOS opens the door to extending mission lifetimes, upgrading payloads, managing orbital traffic, and even reducing space debris.

Figure 1.5: In-Orbit Servicing Mission Concept

At the heart of IOS missions lies the technological challenge of autonomous or semi-autonomous rendezvous and docking. Unlike the International Space Station (ISS), where cooperative docking ports and human crews facilitate servicing, most operational satellites were never designed to be serviced. This means the servicer must be able to capture non-cooperative targets — satellites without docking adapters, sometimes tumbling or passively stable — requiring the use of robotic

arms, vision-based navigation, and sophisticated control algorithms. Once the servicer establishes a stable configuration, it can carry out its primary functions.

One of the most significant functions of IOS is refueling. Satellites in geostationary orbit (GEO), for example, rely heavily on onboard propellant for station-keeping maneuvers. Once this propellant is exhausted, the satellite must be retired, regardless of whether its payload is still fully functional. An IOS spacecraft equipped with a propellant transfer system can dock with the satellite and replenish its fuel tanks, effectively resetting the mission lifetime clock and allowing years of additional service. This capability drastically changes the economics of GEO communication satellites, where revenue is directly tied to operational longevity.

Another key function is repair and maintenance. Damage to spacecraft components, such as a malfunctioning antenna or solar array, can cripple a mission. Through the use of robotic manipulators and modular toolkits, IOS spacecraft can perform corrective actions such as deploying stuck mechanisms, replacing modular parts, or restoring partial functionality to otherwise lost assets. In the future, satellites may even be designed with replaceable "orbital modules," making repairs and upgrades more akin to maintenance cycles on Earth.

A further application of IOS is relocation. Satellites often need to change orbital positions, especially in geostationary orbit where slots are valuable and subject to regulatory coordination. A servicing spacecraft can attach itself to a client satellite and act as a tug, moving it into a new slot, transferring it to a graveyard orbit at the end of life, or repositioning it into medium or low Earth orbit for a new operational phase. This tug capability can also serve military and Earth Observation purposes, where responsive repositioning of assets is highly desirable.

The broader vision of IOS extends beyond simple life-extension missions. It provides the foundation for an in-space economy where spacecraft can be maintained, upgraded, and even assembled on-orbit. Future missions may involve installing new payload modules to enhance functionality, building larger space telescopes piece by piece, or

servicing lunar and Mars communication relays. By reducing waste, extending functionality, and enabling new mission architectures, IOS missions transform satellites from disposable assets into serviceable infrastructure.

1.4 Objectives of the Thesis

The central aim of this research is to create a simulation environment that is characterized by both robustness and flexibility. To achieve this, a set of key design guidelines was established for the project:

- **Compliance with ECSS SMP standard:** Adherence to the European Cooperation for Space Standardization (ECSS) SMP standard ensures that the simulator meets recognized industry benchmarks for quality, interoperability, and reliability. This adherence is fundamental to enabling the reuse of models across different simulation environments and facilitating their exchange between various organizations and missions (Clause 1, Scope). Compliance with this standard establishes a shared understanding of core concepts and a common type system, both of which are essential prerequisites for effective collaboration and model portability. This standardization obviates the need to recode models for disparate simulation platforms, thereby simplifying the integration of new components and fostering partnerships with international stakeholders. The standard, for instance, specifies common time kinds (e.g., `EpochTime`, `MissionTime`) and an object-oriented architecture, ensuring all components speak the same technical language. For this reason the SIMULUS framework uses C++ as its native language, as it is the language of the ECSS-E-ST-40 standard.
- **Integration of SIMULUS** This involves the integration of the SIMULUS to form a comprehensive and dependable system. This aligns with the ECSS-E-ST-40-07C standard, which seeks to enhance the portability and reuse of simulation models by defining a standardized interface between the simulation environment and

its models. The SIMLUS framework provides the foundational communication and data management infrastructure required for their integration. This hybrid approach capitalizes on the strengths of a commercial platform for core architecture and the reliability of publicly-funded, mission-proven components for specific functional models.

- **Creation of a modular, extensible simulation environment:** The architectural design prioritizes modularity, enabling the seamless addition or modification of subsystems, and extensibility, allowing the system to support diverse mission scenarios without necessitating a complete redesign. The SMP architecture is specifically engineered to support this paradigm, as it distinctly separates **Simulation Models** (which provide application-specific behavior, such as the EPS components) from the **Simulation Environment** (which supplies core services like the scheduler and time keeper) (Clause 4.3). The modularity of the system ensures that changes or updates to a single subsystem, like the EPS, do not introduce cascading failures in other, unrelated components, such as the Attitude Control System (ACS). This structural separation simplifies maintenance, allows for parallel development by specialized engineering teams, and ensures the simulation can evolve concurrently with the physical spacecraft's design.

- **Support for various mission scenarios:** This entails the development of a simulator capable of operating under a broad spectrum of operational conditions, from standard maneuvers to complex in-orbit servicing tasks. This functionality includes the capacity to simulate diverse orbital environments, communication latencies, and various hardware states, encompassing both nominal operation and predefined failure modes, to provide a truly comprehensive and realistic testing platform. For example, a mission scenario might involve simulating a rendezvous and docking procedure, requiring precise modeling of both the chaser and target satellites, their propulsion systems, and their relative dynamics. The simulator's design must be robust enough to handle the complexity and

dynamic range of such events.

- **Enables realistic testing, validation, and analysis of spacecraft operations:** This represents the overarching objective, accomplished through the simulator's ability to precisely model the intricate interactions between different subsystems and the space environment. The simulator provides a robust and dependable platform for OBSW validation, mission performance analysis, and personnel training in a risk-free virtual setting. This is crucial for verifying the logic of the OBSW, detecting potential deadlocks, and optimizing operational sequences before they are ever executed on the real satellite, a process that directly impacts mission safety and success.

These guidelines are applied to any simulator developed under the SIMULUS framework, extending beyond the scope of this thesis. The primary focus of this thesis was on the development of the EPS, with the remainder of the simulator to be developed in future work.

Specifically, the core objectives of this thesis were:

- **To establish the development environment for the simulator and the EPS simulator**, encompassing the installation of the SIMULUS SMP framework and the configuration of the development environment. This constitutes a crucial initial step to ensure compliance with the standard's defined interfaces and common type system, which are necessary for component interoperability (Clause 4.2). The absence of this standardized foundation would render communication between components unfeasible, undermining the core principle of a modular design.

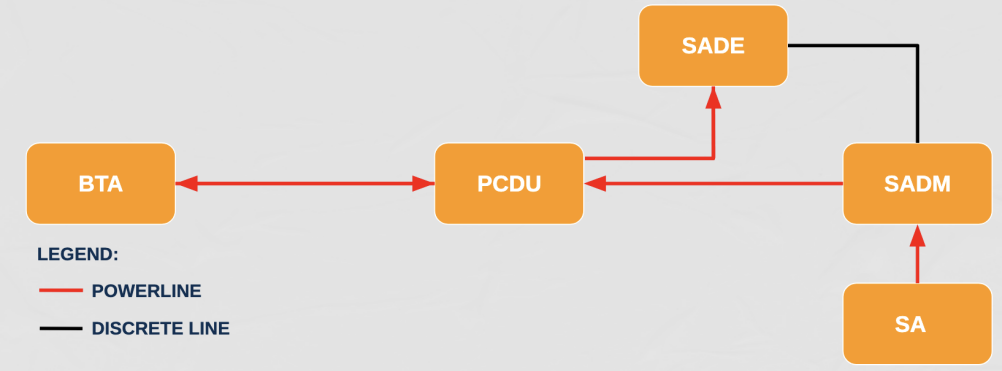

Figure 1.6: EPS Architecture

- **The development of a Battery model** capable of replicating the behavior of a real IOS satellite battery, including charging, discharging, power provision to subsystems, and communication with other components via the SIMULUS framework interfaces. This model must accurately represent the battery's state of charge, temperature, and degradation over time.
- **The development of a Solar Array model** designed to provide power to the satellite subsystems, with the functionality to track the sun cycle and supply power to the entire spacecraft. This model incorporates algorithms to calculate solar flux based on the spacecraft's orientation, a key factor in power generation.
- **The development of a Power Distribution and Control Unit (PCDU) model** capable of managing the power supplied by the sources and distributing it to the various loads as required. According to the ECSS standard, such application-specific elements are referred to as **components**, which are the fundamental building blocks of a simulation, instantiated with a well-defined contract to their environment (Clause 3.2.4). These components serve to encapsulate the behavior of physical hardware, allowing engineers to concentrate on logical interactions rather than low-level implementation details.

- **The integration of SIMULUS framework interfaces** to facilitate communication between components and the simulator. This involved the implementation of MILBUS1553 and PowerLine protocols. Thanks to the compliance with SMP standard, inter-component communication is handled through **three primary methods**: direct interface-based, data flow-based, and event-based communication. These standardized communication methods guarantee that components from different developers or departments can interact effectively. The implementation of specific physical protocols like MILBUS1553 and PowerLine is therefore managed within the framework's communication layer, preserving the platform-independent nature of the higher-level models and allowing for easier model exchange and reuse.
- **Implementation of MILBUS1553 and PowerLine protocols** to enable communication between the EPS simulator and other subsystems. The MILBUS1553 protocol is a widely used standard for spacecraft data bus communication, while the PowerLine protocol is used for power distribution and control. These protocols are essential for ensuring that the EPS can effectively interact with other components of the satellite simulator.

In chapter 4 it will be discussed the physical functionalities of each component and how they have been designed and developed.

Chapter 2

State of the Art

This chapter provides a comprehensive overview of the current state of satellite simulation technology and software engineering practices that are used to provide high-fidelity simulation environments. It describes the technologies used in the development of the thesis, including, UML diagrams, design patterns and the SIMULUS framework. It also discusses some other possible options used for satellite emulator development, highlighting the advantages and disadvantages of each approach.

2.1 Technological Overview

2.1.1 C++ Programming Language

C++ is a high-performance, object-oriented programming language that is widely used in the development of complex systems, including satellite simulators. Its features, such as low-level memory manipulation, strong type checking, and support for both procedural and object-oriented programming paradigms, make it particularly suitable for developing high-fidelity simulators. C++ allows for fine-grained control over system resources, which is essential for simulating the intricate behaviors of satellite subsystems. Also C++ is the native language of the ECSS-E-ST-40 standard, which is used in this thesis.

2.1.2 VMWare Workstation Pro

VMWare Workstation Pro is a powerful virtualization software that allows users to run multiple operating systems on a single physical machine. It was used to create a virtual machine environment for the development of the satellite simulator. It was needed to simulate the operating system beacuse in order to introduce the Telecommunications between the OBC and earth station the simulator needs another windows machine that runs the ECHO environment.

2.1.3 SUSE Linux Enterprise

SUSE Linux Enterprise is a robust and reliable operating system that provides a stable platform for running complex applications, including satellite simulators. It is known for its security features, scalability, and support for enterprise-level applications. The choice of SUSE Linux Enterprise was made to fulfill the requirements of the SIMULUS framework which can be run on any UNIX-like operating system, but is optimized for SUSE Linux Enterprise. This choice ensures compatibility with the framework's requirements and provides a stable environment for development and testing.

2.1.4 UML Diagrams - Magicdraw

UML (Unified Modeling Language) diagrams are a standardized way to visualize the design of a system. They are particularly useful in software engineering for modeling the architecture, components, and interactions within a system. In the context of satellite simulation, UML diagrams can be employed to represent the structure and behavior of the simulator, including its components, their relationships, and the flow of data between them. The software used for creating UML diagrams in this thesis is MagicDraw, a powerful tool that supports various UML diagram types, including class diagrams, sequence diagrams, and activity diagrams. Later on it will be discussed how these diagrams are created and used in the development of the simulator.

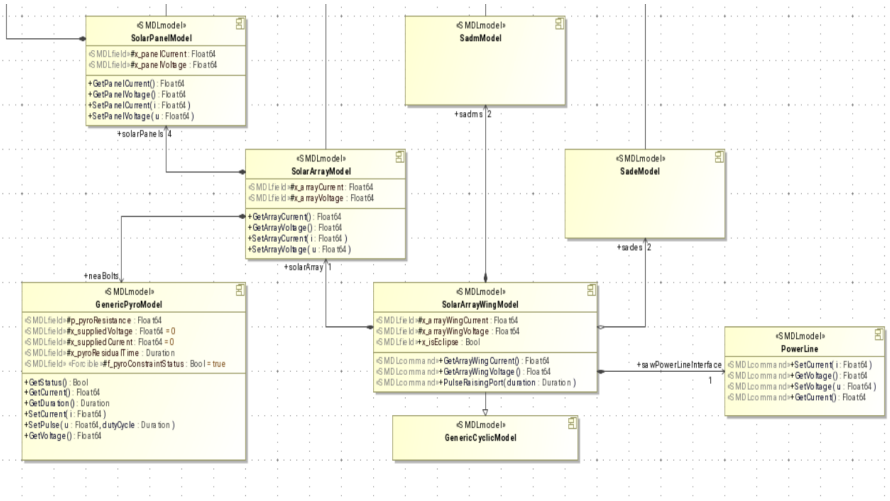

Figure 2.1: MagicDraw UML Diagram Example

2.1.5 SIMSAT

SIMSAT is a specific, object-oriented software architecture for real-time satellite simulation. It's a "simulation kernel" or a core framework that provides the infrastructure for building satellite simulators. SIMSAT was developed by the European Space Operations Centre (ESOC) and is used to create high-fidelity simulators for satellite missions. SIMSAT provides the essential services and interfaces for the simulator, such as scheduling, data management, and control interfaces, allowing developers to focus on creating the specific models for the satellite's subsystems (e.g., power, attitude control, communications). SIMSAT is an example of a simulation infrastructure that can be made SMP-compliant to allow for model reuse.

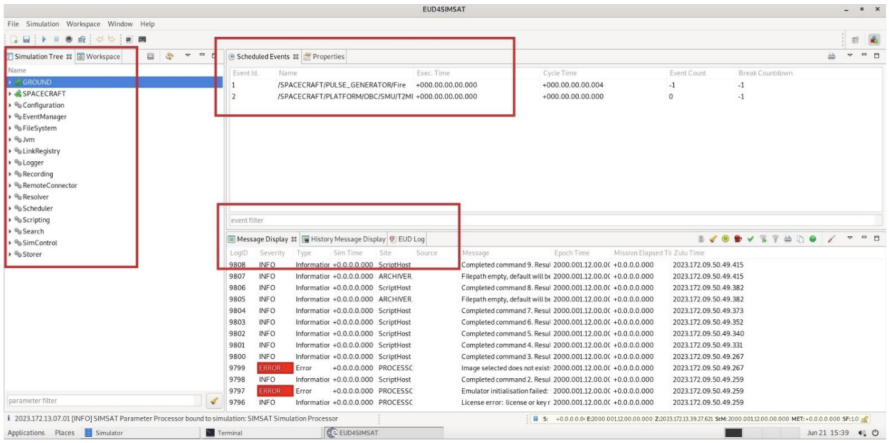

Figure 2.2: SIMSAT Architecture

2.1.6 SIMULUS 10.3

SIMULUS is the European Space Agency's complete simulation infrastructure, essentially a suite of frameworks and libraries that provide the backbone for spacecraft simulators. Unlike SIMSAT, which is just the simulation kernel responsible for time management and model execution, SIMULUS defines the overall architecture, interfaces, and toolchain that allow a simulator to be built, integrated, and connected to external systems. It establishes the standards and runtime services that all models and simulation kernels must adhere to so that different subsystems can interoperate seamlessly.

At its core, SIMULUS provides a generic simulation framework compliant with the ESA-endorsed SMP (Simulation Modelling Platform) standard. This means that subsystem models, whether describing orbital dynamics, thermal behavior, or power management, are implemented in a standardized way with well-defined interfaces for initialization, time stepping, and data exchange. These models are compiled, usually in C or C++, and loaded into the simulation environment as shared libraries. SIMSAT is one possible runtime kernel within SIMULUS that executes these SMP models, but SIMULUS itself encompasses more than just execution: it also handles the definition of model metadata, the configuration of the simulator, the management

of telemetry and telecommand flows, and the connection to ground control systems.

Technically, SIMULUS provides the infrastructure to integrate not only analytical or physics-based models but also processor emulators like TEMU4. TEMU4 runs unmodified satellite flight software binaries, which can then exchange data with subsystem models running under SIMSAT. The SIMULUS framework ensures that timing, synchronization, and data exchange across these heterogeneous components remain consistent. For example, during the simulation of an in-orbit servicing mission, SIMULUS can orchestrate the spacecraft dynamics models (running in SIMSAT), the robotic arm kinematics and contact dynamics libraries, and the onboard flight software executing inside TEMU4. The result is a single coherent simulator that behaves as though an actual spacecraft and its client satellite were in orbit.

Another crucial technical role of SIMULUS is to interface with external systems using standardized telemetry and telecommand protocols, especially the ECSS Packet Utilization Standard (PUS). This allows a simulator built with SIMULUS to connect directly to a mission control system, validating operational procedures under realistic conditions. For IOS missions, this means that ground operators can test how the servicer will respond to docking commands, refueling sequences, or relocation maneuvers, using the exact same control consoles they would operate during a real mission.

In short, SIMULUS can be thought of as the ecosystem within which SIMSAT and TEMU4 operate. It defines the rules of interaction, provides the supporting services for data exchange and control, enforces SMP compliance for portability and reuse of subsystem models, and ensures interoperability with mission control infrastructures. Without SIMULUS, you could still run models in isolation with SIMSAT or emulate software with TEMU4, but you would not have a complete spacecraft simulator capable of supporting development, testing, training, and validation across the full spacecraft lifecycle.

2.1.7 Simulink

Simulink is a MATLAB-based graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in automatic control and digital signal processing for multidomain simulation and model-based design.

2.1.8 ECHO Environment

ECHO Windows machine is a specialized software environement developer by Thales Alenia Space that provides a set of tools and services for developing and testing satellite software that is SIMULUS compliant. It is used to simulate the ground control systems that interact with the satellite and specifically it sends telemetries and telecommands, providing an interface to send raw bytes of data written in hexadecimal format or to standardized standard based on the protocol used in the communication. the TM/TC exchange is managed through ECHO SW while the externally developed models use the ISIS interfaces to connect with the board, trough CAN Bus or the 1553 Bus.

2.1.9 TEMU Processor Emulator

An emulator is a software or hardware system that mimics the behavior of another system, allowing hardware designed for one environment to run in a different one. Specifically for this case the TEMU emulator allows the emulation of the OBC of the satellite. TEMU is a widely used processor emulator that supports a variety of architectures including ERC32, LEON2, LEON3, LEON4, LEON 5, P2020 and ARM.

2.1.10 PUS Protocol

The Packet Utilization Standard (PUS) is a protocol defined by the European Cooperation for Space Standardization (ECSS) for the exchange of the PUS defines how telecommand (TC) packets and telemetry (TM) source packets are used for remote monitoring and control of spacecraft subsystems and payloads. It structures operations into services, each corresponding to fundamental spacecraft functions such as telecommanding, verification, telemetry reporting, on-board scheduling, monitoring, memory management, and troubleshooting. [3]

More of this will be discussed later on during the development of the MILBUS1553 interface.

Chapter 3

Feasibility study

This section of the thesis is dedicated to the feasibility study meaning that it will describe

- How the environment was configured, the tools used for the development of the simulator and the difficulties met during the setup
- The architecture of the simulator and how it is structured
- The technological choices made during the development.

3.1 Development Environment

This section will discuss the development environment used for the simulator, including the tools and technologies employed, as well as the challenges encountered during the setup process. Since the research was conducted in collaboration with Thales Alenia Space the project and the development environment needed to be secured and isolated from the rest of the network. The machine was configured to be working on a private VLAN that allowed communication with other machines in the laboratory and the gateway to the internet, but not with the rest of the network. After this initial configuration it was essential to retrieve the best possible performances from our machines. The computers used for the development were equipped with

- Intel Core Ultra 9 285K Processor
- 128GB RAM
- 2 TB Hard Drive
- 1 x 24" LED monitor
- Video adapter 1 GB for double monitor
- Mouse
- Keyboard
- Dual Ethernet interface

The main problem with this machine was the processor since the Intel Core Ultra 9 285K processor does not allow Hyper-Threading Intel technology. This means that in order to push the emulation environment we needed to assign specific cores called P-Cores of the 24 present to the process that was running the VMWare Workstation Pro software. This was done to ensure that the virtual machine had enough resources to run smoothly and efficiently, as the simulator requires significant computational power to simulate the various subsystems of the satellite.

3.2 Simulator Architecture

To start things off it is important to first describe the architecture of the simulator and how it is structured.

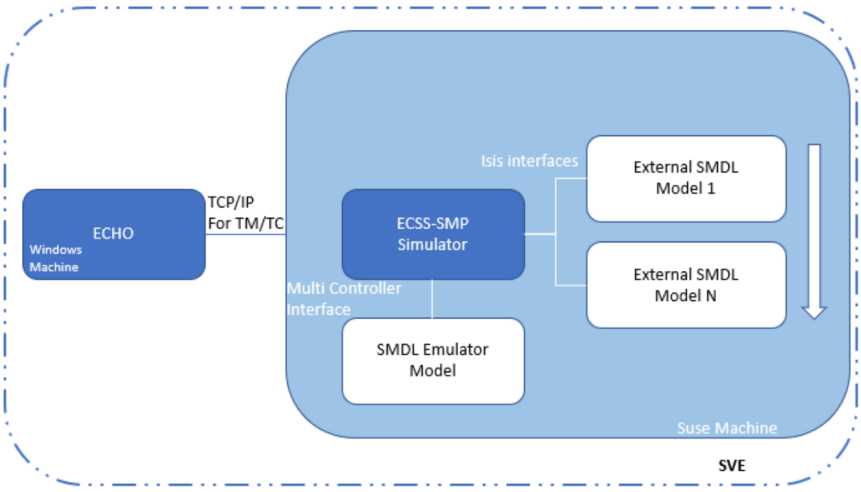

Figure 3.1: Internal components and external interfaces.

As shown in Figure 3.1 and briefly said in Chapter 2, the simulator is composed of several internal components and external interfaces. The main concept is that the machine that runs suse hosts the SIMSAT simulator that provides a GUI for managing and allowing interactions between models while the ECHO environment runs on a separated Windows machine that simulates the ground station control which sends telecommands and recieves telemetries from the satellite. Models are developed in C++ and follow the ECSS-E-ST-40-07C standard, which defines the Simulation Model Portability (SMP) standard. This standard ensures that models can be reused across different simulation platforms, promoting interoperability and efficiency in the development process. This concept is called **model portability** and is a key feature of the SIMULUS framework.

3.3 ECSS SMP Simulator - SIMSAT

SIMSAT is a simulation infrastructure framework developed by the European Space Agency, designed to provide the core environment for spacecraft simulators. It is not itself a simulator of satellites, but rather a simulation kernel that manages how different subsystem models interact, execute, and synchronize during a simulation. SIMSAT provides a

simulation engine that handles time management, either in real-time or accelerated modes, and coordinates the execution of different subsystem models. It also offers a model integration framework where subsystem models, written in languages like C, C++, or Fortran, can be compiled into dynamically linked libraries and loaded into the SIMSAT environment. These models represent specific spacecraft components and are executed under the scheduling of the kernel. SIMSAT also provides control and monitoring facilities such as GUIs, logging systems, telemetry displays, and debugging tools, making it suitable for training operators and validating mission procedures.

3.4 SMDL Emulator Model - TEMU4

Technically, TEMU4 provides instruction-accurate or cycle-approximate emulation of processors commonly used in space applications, such as LEON (SPARC-based), ARM, and other mission-specific architectures. It translates the binary instructions of the flight software into host machine instructions and executes them while also emulating peripherals, memory, and bus systems. This allows the same compiled flight software image that would fly on the satellite to be executed inside TEMU4. Through this mechanism, engineers can test bootloaders, operating systems, device drivers, middleware, and high-level mission applications in a realistic environment without risking expensive hardware.

The tool provides interfaces for simulation control and monitoring, so developers can step through execution, inject faults, or monitor registers, memory, and I/O devices. TEMU4 can integrate with higher-level spacecraft simulators like SIMSAT, creating a full testbed where SIMSAT models the spacecraft subsystems and dynamics, while TEMU4 executes the on-board software stack in a virtual processor environment. This combination allows for end-to-end verification of how software reacts to spacecraft behavior and vice versa.

Chapter 4

Design and Development

In this section it will be discussed the design patterns and the development process of the EPS. As mentioned in the previous sections, work was devided in three main parts including the develpment of the Battery model, the Solar Array model and the PCDU model. The part that will be discused will be the battery model and the development of the PCDU model, the PowerLine interface and the MILBUS1553 interface. To do so it is important to first understand the EPS physical functionalities and how they interact with each other.

The next section will describe the physical functionalities of a prototype of Battery used for space missions and how the design was extracted from them.

Specifically these steps are the ones that will be described in the following section:

- Description of the physical functionalities and lifecycle of a space Battery
- Description of the UML diagrams designed starting from the physical functionalities
- Description of the development of the Battery model, PowerLine and MILBUS1553 interfaces, specifically:
  - XML configuration files and their structure.

– The main classes in the Battery Model and their relationships.
– The main functions of the Battery model and how they interact with each other.
– Development and implementation of the PowerLine interface, discussing some pseudo-code took from the actual code.
– Overview of the MILBUS1553 interface and of the PUS protocol.
– Code implementation of the MILBUS1553 interface and the functions developed.
– Overview on the handling of telecommands and telemetries implemented for the MILBUS1553 interface.

4.1 Space Battery Functionality

A battery is an assembly of interconnected electrochemical cells. Each cell is composed of an anode that provides electrons, a cathode that accepts them, an ion-conducting electrolyte, and a separator to prevent short circuits, all housed within a hermetic case with terminals for the external circuit. Batteries for space missions are divided into two main classes: primary (non-rechargeable) and secondary (rechargeable). Primary cells are ideal for "one-shot" applications, such as long-distance planetary missions, where power is needed for a brief period after a long cruise.

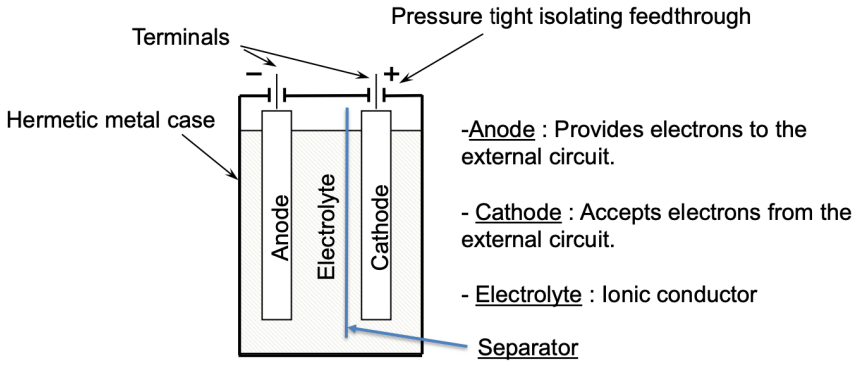

Figure 4.1: Battery Cell [4]

These are typically Lithium (Li) based, chosen for their high specific energy (over 500 Wh/kg), flat discharge characteristic, and very long shelf life of up to 10-20 years. The specific chemistry, such as Li-SOCl2 or Li-MnO2, is selected based on the mission's requirements for energy, power, and operating temperature. Secondary cells are essential for missions requiring repeated charge and discharge cycles, like those in Low Earth Orbit (LEO) or Geostationary Orbit (GEO) that experience frequent eclipses. Currently, Lithium-Ion (Li-Ion) technology is the standard for rechargeable space batteries, offering high specific energy up to 180 Wh/kg and a long cycle life. These cells can be assembled in different configurations, such as connecting serial strings in parallel ("s-p") or parallel groups in series ("p-s"), depending on factors like redundancy and management complexity. The performance of any battery is heavily influenced by its temperature and State of Charge (SoC).

For space applications, batteries are typically designed for an optimal operating range of 0°C to +30°C. Maintaining this temperature is critical; higher temperatures can improve performance by reducing internal resistance but also accelerate irreversible chemical degradation, while low temperatures increase resistance. Thus, thermal management systems with heaters and coolers are often necessary to ensure optimal performance and longevity. Key performance metrics define a battery's capability. Specific Energ(Wh/kg) and Specific Volume (Wh/l) measure its energy density by mass and volume, respectively. Depth of Discharge (DoD) indicates the percentage of energy used per cycle and is a critical factor in determining the battery's lifespan; a higher number of cycles, typical for LEO missions, necessitates a lower DoD, whereas GEO missions with fewer eclipses can tolerate a higher DoD. Proper management is paramount for the safety and reliability of Li-Ion batteries. Charging typically follows a constant current phase until a voltage limit is reached, followed by a taper charge at constant voltage. It is crucial to prevent both overcharge (above 4.1-4.2V per cell) and over-discharge (below 3V per cell), as these conditions can lead to irreversible damage or dangerous thermal runaway events. To mitigate these risks, cells incorporate multiple safety features. These

protections include pressure-sensitive devices to prevent overcharge, "Polyswitch" devices to limit current during an external short-circuit, and multi-layer "shut-down" separators that melt at high temperatures to reduce ionic conductivity. Within the spacecraft's Electrical Power System (EPS), the battery plays a central role. It stores energy from the solar array during sunlight, provides power during eclipses, and supplements the solar array during peak power demands. The Power Control and Distribution Unit (PCDU) manages these functions using a Battery Charge Regulator (BCR) and a Battery Discharge Regulator (BDR). The BCR controls charging and protects against overcharge, while the BDR regulates the power supplied to the spacecraft bus and prevents over-discharge [4].

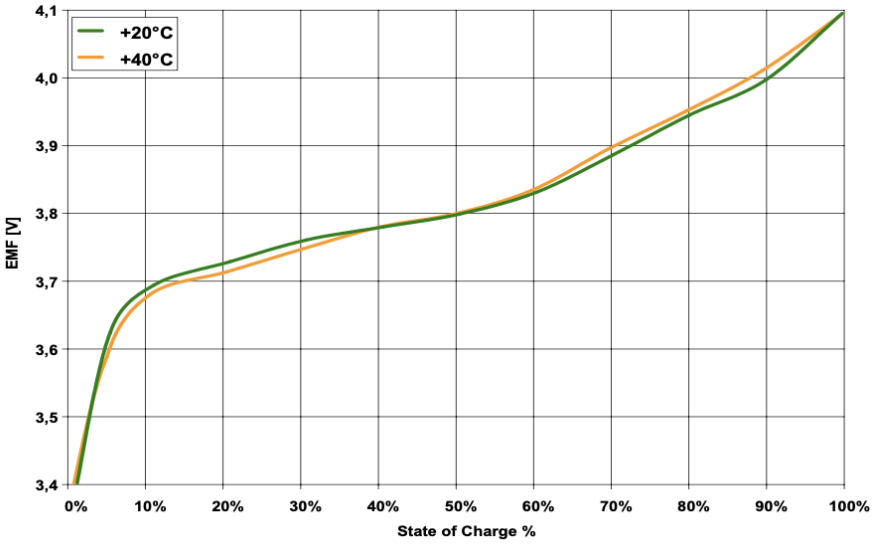

Figure 4.2: State of Charge

This image shows the relationship between the State of Charge (SoC) and the Elecromotive Forse (EMF) of a Li-Ion battery. As is easy to imagine the bigger the SoC the higher the EMF of the battery.

To simulate the correct behavior of a battery it was important to first underline the correct specifics of the IOS satellite battery. For privacy and security reasons these informations cannot be disclosed within this thesis. However, the specifics and the functionalities modeled are

representative of the real IOS battery.

4.1.1 Battery UML Design

The engineering design process requires to extract the software requirements from the physical functionalities of the system. The battery required to carefully read the documentation provided by Thales Alenia Space and understand the architecture of the battery. For IOS case the battery is composed of Battery Cells connected in parallel, each cell has a specific capacity, voltage, internal resistance and a maximum charge and discharge current. The battery is connected to the PCDU though a single Power Line interface that allows to transfer current between the two components. Redundancy was not considered during the development of the battery model, although it is present in the real IOS satellite battery.

The first step for a correct design, considering that the ECSS SMP standard is very similar to a factory method design pattern, is to create an UML diagram. As mentioned during the State of the art chapter, UML diagrams were created using the MagicDraw software. To start things off it was important to understand the environment of the SIMULUS standard associated with MagicDraw. MagicDraw allows to define custom profiles suitable for ECSS standard, the preset was already available in the development environment provided and it was only required to understand how to create the UML diagrams.

Following the UML of the battery model:

The UML diagram is structured as follows:

- BtaModel: this is the main class of the battery model, it inherits from the SMDLmodel and GenericCyclicModel classes provided by the SIMULUS framework. It contains all the attributes and methods necessary to simulate the battery behavior, including state of charge, voltage, current, temperature, and methods for charging, discharging, and updating the battery state.

- BtaCellPack: this class represents a pack of battery cells connected in parallel. It contains attributes such as the number of cells, total

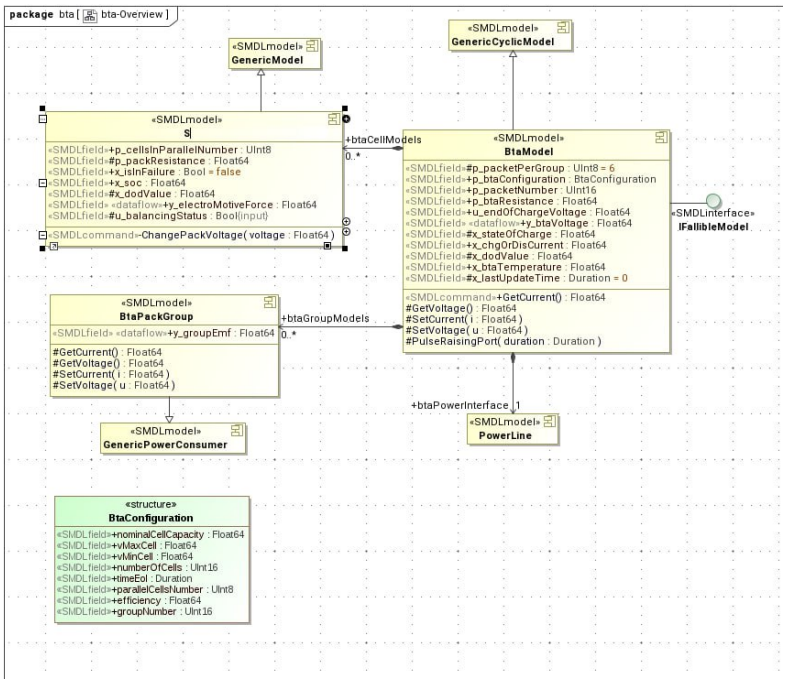

Figure 4.3: Battery UML Diagram

capacity, and methods to calculate the overall voltage and current of the pack. Also methods inside this classes are called by the BtaModel class to regurally update the state of the pack.

- BtaPackGroup: this class was created to represent a group of battery cell packs. Originaly it was implemented to manage redundancy and failure of the packs but this functionality is not still implemented and will be added in future work.

- BtaConfiguration: this structure contains all the configuration parameters for the battery model, also defines the numbers for the first time the battery is turned on.

- PowerLine: finally the PowerLine interface is a standard interface provided by the SIMULUS framework, this was defined but not initialized in the UML diagram. The initialization was performed later on during the coding of the battery model.

As shown in Figure 4.3, the battery inherits from the `SMDLmodel` GenericModel and GenericCyclicModel, this is because for the ECSS SMP standard models are required to inherit from a hierarchical structure of classes that provide functionalities for generic models. Specifically, a battery, is required to be constantly running and updating its state, therefore it was chosen to inherint from the `GenericCyclicModel` class. For SIMULUS, but in general for C++ programming, there are some methods that are mandatory to implement in order to have a working model. Choosing the correct model to simulate the battery is a crucial step, as it defines the behavior and interaction of the battery within the simulation environment.

Once the UML diagram and design was completed and agreed on with the supervisor, MagicDraw allows to export the UML diagram in a C++ code skeleton. This skeleton provides the basic structure of the classes and methods defined in the UML diagram,without the actual implementation of the methods. This skeleton was then used as a starting point for the development of the battery model, filling in the methods with the actual logic and algorithms to simulate the battery behavior. The skeleton looks like something like this:

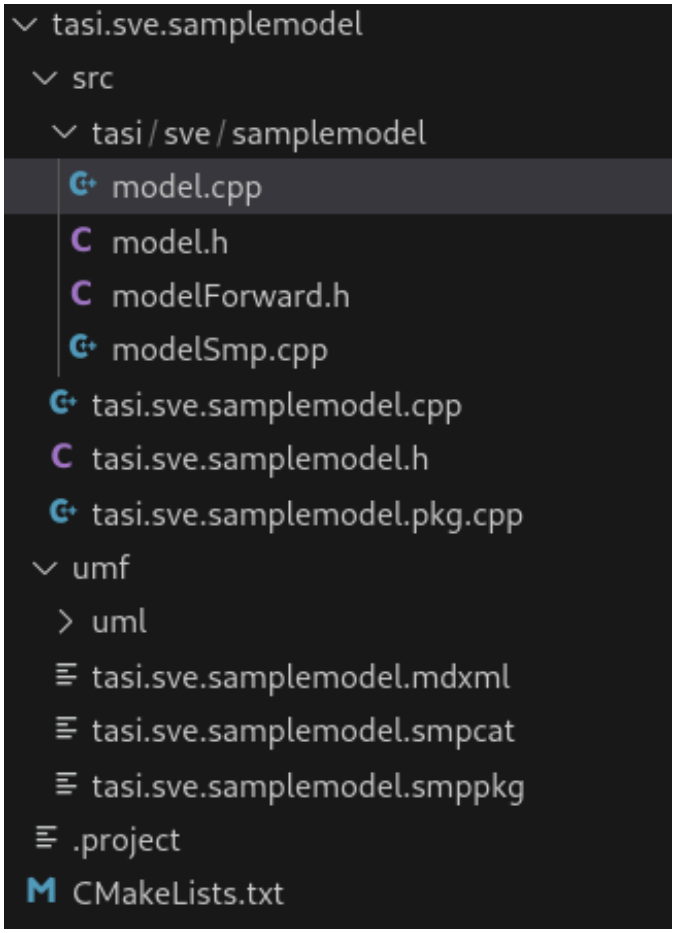

Figure 4.4: Sample Code Skeleton

The code skeleton also includes the import of the necessary libraries and namespaces required for the model based on the model from which the specific class inherits. This ensures that all the required dependencies are available for the implementation of the battery model. It is also worth mentioning that this is not the only way to create a model for SIMULUS, it is also possible to manually do it by creating the classes, importing the right libraries and inheriting from the right classes. As it will be discussed later on, some attributes and methods were added manually without passing through the UML design first. The methods shown in the UML diagram are the ones that were imagined being essential at the beginning of the design process, but during the development some methods were added or removed based on the actual

needs of the model.

4.2 Battery Model Development

This section is one of the core arguments of the thesis, as it will describe the development process, and choices that were made during the implementation of the battery model. As explained in the previous section, the starting point was the code skeleton provided by MagicDraw, so the first idea that came to mind was to start implementing the methods one by one. But first it was essential to understand how SIMSAT works and how to test the model, so that it would have been possible to test the methods as they were implemented. SIMSAT connects to the code through XML files that define the models and their configuration. Specifically for the configuration that was provided by Thales Alenia Space, the core XML file merged all the models that needed to be shown in the SIMSAT GUI like so:

```
<SimpleNode TypeName="tasi::sve::common::CommonRoot" Name="STUBS" Container="Models">
    <Description>TODO</Description>
    <SubNodes>rcu</SubNodes>
    <SubNodes>bta</SubNodes>
    <SubNodes>saw</SubNodes>
    <SubNodes>pcdu</SubNodes>
    <SubNodes>samplemodel1</SubNodes>
    <InstanceNode TypeName="tasi::sve::dummy::spacewire::SpwDummyNode" Name="SPW_DUMMY_1_N" Container ="Models">
        <Description>TODO</Description>
    </InstanceNode>
    <InstanceNode TypeName="tasi::sve::dummy::spacewire::SpwDummyNode" Name="SPW_DUMMY_2_N" Container ="Models">
        <Description>TODO</Description>
    </InstanceNode>
```

Figure 4.5: XML Configuration File

This XML file defines the models that will be loaded into the SIMSAT environment, including their names, types, and configuration files. The `<SubNodes>` refere to aliases creaded in the original BTA.xml file that needed to be created manually. The following image shows an example of XML file since it is not possible to show the original one.

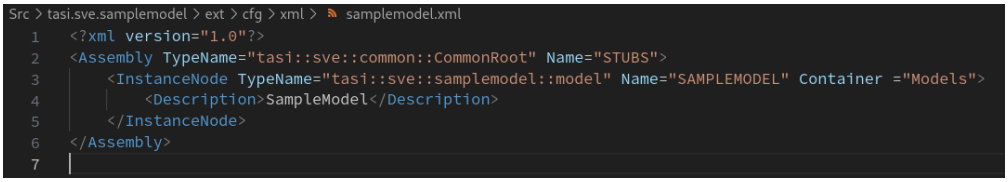

Figure 4.6: Sample XML Configuration File

Once opened the SIMSAT GUI by launching the script `./SIMSAT.sh` in the terminal, the structure of the satellite and the SIMULUS environement is visible. It is possible to observate the fields previously created in the UML diagram and their values. Once SIMSAT is running and correctly configured with the XML files showing the skeleton of the battery model, it was then possible to start implementing the methods.

4.2.1 BtaModel Class

The class BtaModel once initialized calls a method called `CreateAllContainedModels(Smp::ISimulator* dynsim)` which takes as input a pointer to the a Simulator object. This method is used to create all the sub-models that are contained within the battery model. First of all it calls a function called `BtaXmlFilParser()` which based on the BtaConfiguration and the parameters set for `numberOfCells, vMaxCell, vMaxCell, ...` instantiates and calculates the BtaCellPack values for the configuration based on this formula:

$$p_{\text{packetNumber}} = \frac{N_{\text{cells}}}{N_{\text{parallel}}}$$

$$v_{\text{max}} = v_{\text{max}}^{\text{cell}} \cdot p_{\text{packetNumber}}$$

$$v_{\text{min}} = v_{\text{min}}^{\text{cell}} \cdot p_{\text{packetNumber}}$$

After calculating the values for each pack, the method `CreateAllContainedModels` creates the packs in a loop that recursvely calls the method `CreateContainedModel`
which is a method provided by the SIMULUS framework, present in the

CommonRoot namespace, which takes as input the Simulator pointer, the cell name, a pointer to the parent model, and the UUID identifier of the model created. After this each pack field is initiated with the values present in the basic configuration by following the coding practices that avoid any cybersecurity risk. Each input is validated and checked before being assigned to the attribute. The loops iterates for each cell pack present in the battery.

4.2.2 Pulse Raising Port

The first implementation was the `PulseRaisingPort` method. This method is called when the model is started and it is
used to initialize the model. It is present in the `GenericCyclicModel` class and it is mandatory to implement it. In order to implement the `PulseRaisingPort` method it was defined a function called

`UpdateModel(tasi::sve::bta::btaconfig::BtaVariableUpdate variable)`. This function takes as input an enum variable that is used in a switch case to update the model based on the variable passed as input. The cases are the following and for each case a specific logic is defined:

1. BTA_ACTIVE_PACKS: Calculates the number of active packs based on the current state of the battery and update the capacity of the Battery based on the number of active packs.
2. BTA_STATE_OF_CHARGE: Calculates the State of Charge (SoC) for each cell pack using this formula:

$$(v_{btaVoltage} = v_{\text{btaVoltage}} + EMF_{\text{cellPack}})$$

3. BTA_CURRENT: Calculates the current flowing through the battery based on the power line interface and update the current attribute of the battery.
4. BTA_EOC_WORKING_POINT: this switch case is very complex so it requires a separate section to be explained and discussed.

4.2.3 End of Charge Working Point switch case

This switch case is the most complex one because it requires to costantly update both the Depth of Discharge (DoD) of the battery and the SoC of each single cell pack. First the running DoD is saved in a temporary variable, then the Maximum and Minimum voltage is assigned based on the configuration file for each cell. It is important to underline that the current flowing in the battery is represented by only one variable `iChgOrDchgEfficiency`. This variable is positive when the battery is taking power from the PCDU and negative when the battery is giving power to the PCDU. After the assignment of the DoD, if the efficiency is different from zero, the current is set to 0.0 and, if positive the efficiency of the flowing for the current is calculated by following this forumla

$$\text{CurrentEfficiency} = \frac{-\ current}{batteryEfficiency}$$

while if the current is negative the efficiency is calculated by following this formula:

$$\text{CurrentEfficiency} = -\ current \cdot batteryEfficiency$$

Later the DoD is updated by following this formula:

$$\text{tempDod} \mathrel{+}= \frac{i_{\text{ChgOrDchgEfficiency}} \cdot \Delta t}{\text{FROM}_{H \to NS} \cdot C_{\text{actual}}}$$

After this each cell pack is recursively updated by calling a method present in the `BtaCellPack.cpp` class. This method will be discussed in the section dedicated to the BtaCellPack class.

4.2.4 PulseRaisingPort XML Configuration

Once the function `PulseRaisingPort` has been implemented it is important to modify the XML configuration file and define the frequency at which the function will be called. This is done by adding the following line in the XML file:

```
<InterfaceLink Name="">
    <ProviderNode>BTA</ProviderNode>
    <Reference>refClockFrequency_8</Reference>
    <ConsumerNode>PULSE_GENERATOR</ConsumerNode>
</InterfaceLink>
```

This line defines a link between the BTA model and a pulse generator function, which will call the `PulseRaisingPort` method at a frequency of `8 Hz`. This means that the method will be called 8 times per second, allowing the battery model to update its state and perform any necessary calculations.

Lastly the BtaModel.cpp class implements the instantiation of the PowerLine interface by calling the method `SetPortImplementation()` which binds by using the keyword `bind` of the std library to the methods:

- GetCurrent: returns the current flowing through the battery.
- GetVoltage: returns the voltage across the battery.
- SetCurrent: sets the current flowing through the battery.
- SetVoltage: sets the voltage across the battery.

This initialization is done by assigning the `powerPortImplementation.<method><Current/Voltage>Port` to the corresponding method. This section will be largerly discussed in the PowerLine section.

4.2.5 BtaCellPack Class

The BtaCellPack class represents a pack of battery cells connected in parallel. It contains attributes such as:

- packResistance: represents the resistance of the cell pack.
- dodValue: represents the Depth of Discharge (DoD) of the cell pack.

- balancingStatus: represents the balancing status of the cell pack.
- electromotiveForce: represents the electromotive force (EMF) of the cell pack.
- isInFailure: boolean that indicates if the cell pack is in failure.

It also implements methods that are called by the BtaModel class to update the pack recursively.
The `UpdateBtaPack(Smp::Float64 iBta)` method takes as input the current flowing in the battery and calculates the current dodValue of the cell pack by following this formula:

$$x_{\text{DoD}} \mathrel{+}= \frac{i_{\text{Bta}} \cdot \Delta t}{\text{FROM}_{H \to NS} \cdot \left(C_{\text{nominal}} \cdot N_{\text{parallel}}\right)}$$

The Depth of Discharge (DoD) of a battery is defined as the fraction of the nominal capacity that has been discharged up to time t:

$$\text{DoD}(t) \;=\; \frac{1}{C_{\text{total}}} \int_0^t I(\tau)\, d\tau$$

where

- $I(\tau)$ is the battery current (positive for discharge),
- $C_{\text{total}} = C_{\text{nominal}} \cdot N_{\text{parallel}}$ is the effective nominal capacity of the pack,
- the integral computes the total charge withdrawn from the battery.

In discrete form, with a simulation time step Δt, the DoD can be updated incrementally as:

$$\text{DoD}_{k+1} = \text{DoD}_k + \frac{I_k \cdot \Delta t}{C_{\text{total}}}$$

Since Δt is typically expressed in nanoseconds while capacity is given in ampere-hours, a unit conversion factor $\text{FROM}_{H \to NS}$ is introduced to ensure dimensional consistency:

$$\mathrm{DoD}_{k+1} = \mathrm{DoD}_k + \frac{I_k \cdot \Delta t}{\mathrm{FROM}_{H \to NS} \cdot C_{\mathrm{nominal}} \cdot N_{\mathrm{parallel}}}$$

This equation corresponds directly to the implementation:

$$\texttt{x\_dodValue} \mathrel{+}= \frac{i_{\mathrm{Bta}} \cdot \Delta t}{\mathrm{FROM}_{H \to NS} \cdot (C_{\mathrm{nominal}} \cdot N_{\mathrm{parallel}})}$$

which updates the state variable `x_dodValue` by accumulating the fraction of capacity discharged in each simulation step.

After updating the DoD, the method computes the EMF of the cell pack based on the updated DoD and the vMin and vMax of the battery by following this formula:

$$EMF = \left(V_{\mathrm{min}} - V_{\mathrm{max}}\right) \cdot x_{\mathrm{DoD}} \; + \; V_{\mathrm{max}}$$

where:

- E is the electromotive force (variable `y_electroMotiveForce`),
- V_{max} (`vMaxCell`) is the maximum cell voltage at full charge,
- V_{min} (`vMinCell`) is the minimum cell voltage at full discharge,
- x_{DoD} (`x_dodValue`) is the depth of discharge, normalized between 0 and 1.

This linear formulation has the following properties:

$$\begin{aligned} x_{\mathrm{DoD}} = 0 \quad &\Rightarrow \quad E = V_{\mathrm{max}}, \quad \text{battery fully charged,} \\ x_{\mathrm{DoD}} = 1 \quad &\Rightarrow \quad E = V_{\mathrm{min}}, \quad \text{battery fully discharged.} \end{aligned}$$

Hence, the equation performs a linear interpolation between the maximum and minimum cell voltage as the battery is discharged. Although the actual voltage–DoD characteristic of real batteries is nonlinear, this linear approximation ensures:

1. computational simplicity for simulations,
2. EMF values always constrained within the physical bounds $[V_{\mathrm{min}}, V_{\mathrm{max}}]$,

3. sufficient accuracy for system-level studies and control design when high-fidelity electrochemical modelling is not required.

This makes the relation a practical first-order model of the EMF as a function of the depth of discharge.

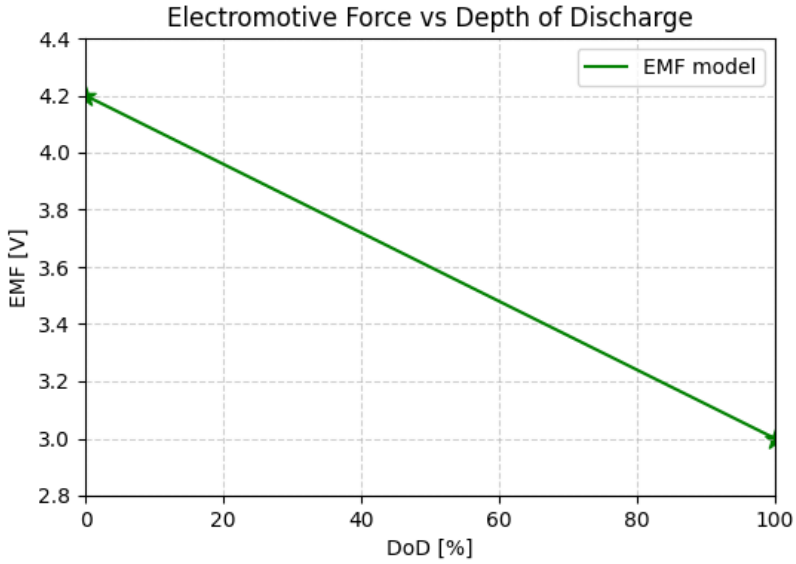

Figure 4.7: Battery EMF mock representation as a function of DoD

This image shows a mock representation of the EMF as a function of the DoD. As shown in the image, when the battery is fully charged the EMF is equal to the maximum voltage of the battery, while when the battery is fully discharged the EMF is equal to the minimum voltage of the battery.

Finally the State of Charge (SoC) is calculated by following this formula:

$$x_{\text{SoC}} = \frac{E - V_{\text{min}}}{V_{\text{max}} - V_{\text{min}}}$$

This formulation ensures the following boundary conditions:

$$\begin{aligned} E = V_{\text{max}} &\Rightarrow x_{\text{SoC}} = 1 \quad \text{(fully charged)}, \\ E = V_{\text{min}} &\Rightarrow x_{\text{SoC}} = 0 \quad \text{(fully discharged)}. \end{aligned}$$

For intermediate voltages $E \in (V_{\min}, V_{\max})$, the SoC takes a value in the range (0,1), thus providing a continuous measure of the remaining charge. The equation can therefore be interpreted as a linear mapping from the physical voltage domain $[V_{\min}, V_{\max}]$ to the normalized SoC domain [0,1].

Although real battery voltage–SoC characteristics are generally non-linear, this linear approximation is sufficient for high-level modeling and system simulations, as it guarantees simplicity, boundedness, and a direct relation between voltage and state of charge.

This functions are called recursively by the BtaModel class to update the state of each cell pack.

4.3 PowerLine Interface Development

This section is dedicated to the development of the PowerLine interface. The PowerLine interface is a standard interface provided by the SIMULUS framework which allows to simulate the power transfer between two components. For this context the components that needed to be connected were the PCDU and the Battery. For doing so it was important to underline and understand which one is the Consumer Node and which one is the Producer node. The steps for implementing a PowerLine interface are the following:

- Define the PowerLine interface in the UML diagram, this step was done in the BtaModel UML diagram as shown in the previous section.
- Enstablish the connection between the two models in the XML configuration file of the Bta by defining the <ConsumerNode>
- Define the methods that will be used to get and set the current and voltage of the PowerLine interface.
- Bind the methods to the PowerLine interface by using the `bind()` method provided by the std library.

- Call the ConnectPort method provided by SIMULUS framework to connect the PowerLine interface to the model. This function takes as input the name of the PowerLine, the PowerLine pointer and a boolean that indicates if the connection is bidirectional or not.

Once the methods have been defined and implemented, in the Pcdu class it was required to initialize and define the PowerLineInterface by:

```
1. Get the list of components connected to the battery \\
power port
2. Select the first component in that list
3. Check if this component implements the \\
IPowerLineIF interface
   a. If yes, assign it to _btmInterface
   b. If no, assign null to _btmInterface
```

Once the dynamic cast succeeds and the pointer _btmInterface is assigned, the BtaModel class effectively gains access to the set of methods exposed by the IPowerLineIF interface. This is a critical step because it transforms the relationship between the battery model (BtaModel) and the power distribution unit (Pcdu) from a loose coupling (simply being connected through a generic port) into a strongly typed, interface-driven interaction. In practice, the Pcdu class implements the IPowerLineIF interface, which means it provides concrete implementations for all methods declared in the interface. By storing a pointer of type IPowerLineIF* in _btmInterface, the BtaModel does not need to know the exact type or internal structure of the connected component. Instead, it can simply call the interface methods, relying on polymorphism to ensure that the appropriate Pcdu implementation is executed at runtime.

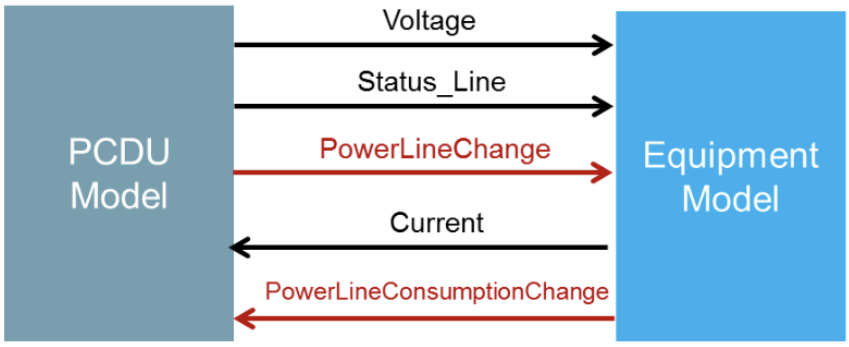

Figure 4.8: Example of a Power Line interface between a power supplier model and a power consumer mode [5]

This design pattern has several important implications. First, it enforces abstraction and modularity. The BtaModel interacts only with an abstract contract (IPowerLineIF), not with the full Pcdu class. This makes the battery model independent of the specific implementation details of the power distribution unit. If in a future evolution of the system the Pcdu class were replaced by another component that also implements IPowerLineIF, the BtaModel code would not require any modification.

Second, this approach provides flexibility and extensibility. Because the cast is performed at runtime using dynamic_cast, the system checks whether the connected component actually supports the interface. If it does, the cast succeeds and the interaction becomes available. If it does not, _btmInterface remains null, and the BtaModel can either refrain from making calls or raise an appropriate error. This means the same software architecture can support different configurations of connected components without hardcoding assumptions into the BtaModel.

Third, binding the Pcdu methods to the interface ensures safe and structured communication between subsystems. The Pcdu exposes only the operations defined in the IPowerLineIF interface, which acts as a contract specifying exactly what the BtaModel is allowed to do. For example, the interface may provide methods to request power, report consumption, or enable and disable certain lines. By restricting access

to these well-defined operations, the architecture reduces the risk of unintended side effects and enforces discipline in subsystem interactions.

From a broader system engineering perspective, this mechanism exemplifies separation of concerns. The battery model is responsible for tracking state-of-charge, depth-of-discharge, and electromotive force, while the power distribution unit is responsible for delivering electrical energy to spacecraft subsystems. The interface provides the bridge between the two domains, enabling the battery to influence and be influenced by the power distribution logic, while keeping each component isolated in its own domain of responsibility.

Finally, in simulation environments such as those used for spacecraft power system verification, this approach is particularly valuable. By abstracting interactions through interfaces, the simulation framework can easily swap in different models of the Pcdu (e.g., a simple stub for early validation, a high-fidelity electrical model for hardware-in-the-loop tests, or even a real hardware connection) without modifying the BtaModel. This ability to decouple the model from the implementation ensures that the simulation remains scalable, maintainable, and adaptable to evolving mission requirements.

4.4 MILBUS1553 Interface Development

4.4.1 MILBUS1553 Overview

For the MILBUS1553 development it was first important to understand how the MILBUS1553 works and what are its functionalities. The MILBUS1553 is a military standard that defines a serial data bus used in avionics and aerospace applications for communication between various subsystems. MIL-STD-1553 is a military standard that defines the mechanical, electrical, and protocol characteristics for a serial data bus used primarily in avionics systems. It specifies a digital, command/response, time-division multiplexing data bus for interconnecting multiple remote terminals (up to 31) with a single bus controller over a dual-redundant twisted shielded pair transmission line. The protocol uses Manchester II bi-phase encoding at a 1 Mbps bit rate

and has fault-tolerant features with dual redundant buses for reliability. [6]

Key features of the MIL-STD-1553 protocol include:

- A single Bus Controller (BC) initiates all data communication with Remote Terminals (RT).
- Messages consist of 16-bit data words with command, status, and data types.
- Data transfers follow a command/response protocol format.
- The bus uses a balanced line physical layer with Manchester encoded data.
- Redundant buses provide increased fault tolerance.
- Defined timing behavior allows RTs to respond within strict timing windows.
- Bus Monitors can passively observe bus traffic without communication participation.

In the IOS mission, MILBUS1553 protocol is used to communicate between the On-Board Computer (OBC) and various subsystems, including the Electrical Power System (EPS). Both the SADE and the PCDU have MILBUS1553 interfaces to receive commands and telemetry data. The OBC acts as the Bus Controller, while the SADE and the PCDU function as Remote Terminals. The Data Packet arrives at the OBC from ground via the TT&C system in a PUS format. As anticipated in the State of the Art chapter, the PUS standard is a protocol for telemetry and telecommand data exchange between ground and space systems. PUS defines a common language of services and packets for controlling spacecraft subsystems. Ground sends standardized telecommand packets, spacecraft replies with telemetry according to the same service definitions. This guarantees reliable, mission-independent operations.

4.4.2 PUS Overview

For simulation purposes the services used is the second one, which is the telemetry service. The telemetry service allows to send telemetry data incapsulated inside a PUS packet. This allows the OBC to automatically decode the data and extract the custom MILBUS1553 command and data word.

The service 2 PUS packet structure is as follows:

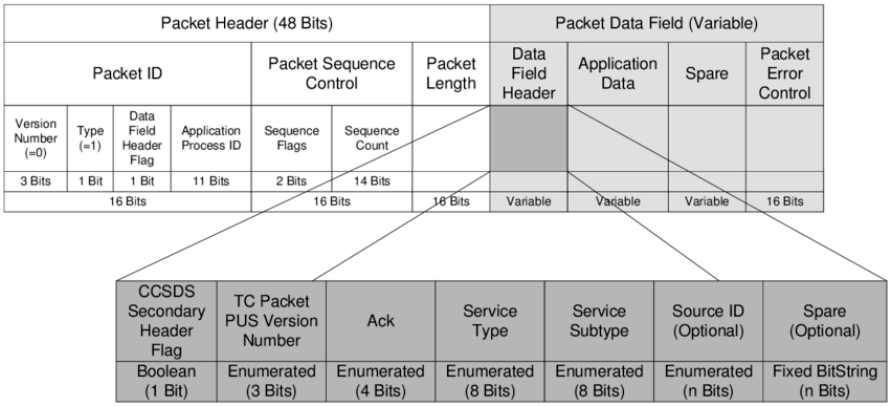

Figure 4.9: PUS Service 2 Packet Structure [7]

As shown in the image, for the simulation case, the MILBUS1553 command and data word are incapsulated inside the Data field of the PUS packet. Usually the defined service is understood by the OBC and an appropriate response is given but for the simulation it was necessary for testing purposes to create a custom service that would allow to send the MILBUS1553 command and data word directly to the PCDU.

4.5 Implementation of the MILBUS1553 Interface

This section describes in detail the implementation of the MILBUS1553 interface within the simulator. The interface was realized through the creation of the class `Pcdu1553.cpp`, which is responsible for managing the communication between the Power Control and Distribution Unit

(PCDU) and the On-Board Computer (OBC) over the MILBUS1553 protocol.

The `Pcdu1553.cpp` class inherits from the `SMDLmodel` and `GenericCyclicModel` base classes, both of which are provided by the SIMULUS framework. SIMULUS offers a standardized interface for MILBUS1553 communication, which automatically generates a set of default communication functions. While this approach provides a generic interface, it also introduces limitations, since the framework does not allow customization of the protocol's behavior. To overcome this restriction, a custom MILBUS1553 implementation was required, tailored to the needs of the simulator.

4.5.1 Interface Functions

The SIMULUS framework requires the implementation of two main functions for bus communication: `BTtoRT` and `RTtoBC`. However, in the current state of the simulator, only the `BCtoRT` function has been implemented and customized, as it represents the most relevant communication direction for the PCDU use case.

The function prototype is defined in the library as follows:

```
virtual void BCtoRT(const ::Smp::UInt16 command,
    ::Isis::SystemIF::DataWords data,
    ::Isis::SystemIF::ExchangeStatus* exchangeStatus,
    const ::Smp::DateTime exchangeEpochTime,
    const ::Smp::Duration exchangeSimTime);
```

The `BCtoRT` function receives as input:

- the complete data word transmitted over the bus,
- the command word,
- a pointer to the exchange status,
- the epoch time of the exchange, and
- the simulation time.

While not all of these parameters are fully exploited in the current version of the simulator, they have been included to ensure scalability and to facilitate future extensions.

4.5.2 Command Handling

At runtime, the function retrieves the latest message received by accessing the `lastMessageReceived` field. This variable is automatically updated based on communications with the OBCs of SIMSAT and TEMU4. Once the message is available, the function executes the appropriate command by evaluating it within a `switch-case` structure.

The commands currently supported by the system are the following:

- `M1553MsgType::CMD_BROADCAST_BC2RT_READ` (for broadcast communications),
- `M1553MsgType::CMD_BC2RT_READ` (for point-to-point communications),
- `M1553MsgType::MODE_BROADCAST_BC2RT_READ` (for mode commands).

Each command triggers a specific behavior. Among them, the `CMD_BC2RT_READ` command is the most complex, since it requires decoding the incoming data word, extracting subcommands, and executing the corresponding action. This process is encapsulated in the function `caseCmdBcRtRead(data, commandW)`, which takes the data word and the command word as inputs.

4.5.3 Subcommand Decoding

The `caseCmdBcRtRead` function is responsible for parsing the data word and identifying the operation to be executed. The decoding logic can be summarized as follows:

```
cmd         = lower 7 bits of data
turnOnOrOff = bit 7 of data
load        = bits 8-15 of data
```

```
switch (commandW.subAddress):
    case SA_2:
        Log("Inside PCDU SA_2")

        switch (load):
            case 0:
                // Control the power state of a specific load
                SetStatusLoad(load, turnOnOrOff)
        end switch
        break

    default:
        Log("Inside PCDU DEFAULT SA")
        break
end switch
```

In this logic:

- the lower 7 bits of the data word encode the command identifier,
- the 7th bit specifies whether a given load should be turned *on* or *off*, and
- bits 8 to 15 identify the target load within the PCDU.

The `subAddress` field of the command word determines which subsystem or functionality of the PCDU is being targeted. For example, `SA_2` (Subaddress 2) is associated with the control of power loads. Within this subaddress, a second switch-case evaluates the `load` variable, and depending on the value of `turnOnOrOff`, the corresponding load is either enabled or disabled.

This hierarchical decision structure makes the implementation modular and scalable: additional subaddresses and load cases can be introduced without altering the core function design. In this way, the simulator can gradually expand to support more complex behaviors of the PCDU, while maintaining a clear mapping between MILBUS1553 messages and PCDU operations.

Chapter 5

Validation and Testing

In this chapter, the validation and testing procedures for the developed battery model and MILBUS1553 interface are discussed. and shown. It is important to understand that the validation and testing of the models are not performed on a real scale since the real data of the IOS mission cannot be shared due to confidentiality reasons. The battery model will be tested by starting to gather data from the SIMSAT GUI and appending it to a text file. This data will be crunched and plotted using Python scripts that provide a visual representation of the battery behavior. The main parameters that will be plotted are the State of Charge (SoC), Depth of Discharge (DoD), Voltage, Current and Power.

5.1 Battery Model Testing

The battery testing is performed by running the SIMSAT environment and launching the BTA model. Once the model is running, the relevant parameters are monitored and recorded over time. Also the simulation stages are checked constantly meaning that the satellite switches from Eclipse, sunlight and Battery only modes. The main parameters that are monitored during the test are:

- State of Charge (SoC) over seconds
- Battery current over seconds

- VBus voltage over seconds

The VBus is the value that indicates the voltage read by the PCDU on the power line interface. This value is calculated as follows:
$VBus = V_{battery} + R_{equivalent} \cdot I_{battery}$
The current could be positive or negative depending on the direction. The logic behind these test is the following:

- The simulator is kept up for a long time, considering that times in the simulator runs 4 times faster than real time, this means that 1 hour of simulation is equal to 15 minutes of real time.
- While the simulator is running the data is gethered and appended periodically, every 0.125 seconds, to some text files that record each step of the simulation, capturing data from both battery and solar arrya.
- The simulation constantly alternates between sunlight, eclipse and battery only modes. This is done to test the battery behavior in all the possible scenarios.
- Finally once the simulations automatically ends, this is done by checking that the simulation time has reached a predefined limit, the data is processed and plotted using a Python script that reads the text files and plots the relevant parameters.

The idea is to recreate as much as possible the real graphs shown in the IOS Power Budget document. These graphs cannot be disclosed due to confidentiality reasons but the main idea is to recreate the same behavior shown in the document.

5.2 Graphs and Results

This section will discuss the results obtained from the testing and the plots generated by the Python script. The main idea is to show that the battery model behaves as expected under various scenarios.

All the graphs are divided in the phases that represent various stages of the simulation which are:

- Phases 1 and 2: starting SOC will be equal to 94% corresponding to PCDU EOCV 61.5
- Phases 2b, 3 and 4: starting in sunlight with the minimum admissible 45% SOC reached at the end of the ascent phase

Specifically:

- Phase 1: Ground and launch
- Phase 2: Launcher Separation to SA deployment
- Phase 2b: SA deployment to target release - starting in sunlight
- Phase 3: In-orbit phase (S-band Tx only)
- Phase 4: In-orbit phase (S-band + X-band Tx)

5.2.1 State of Charge (SoC) over seconds

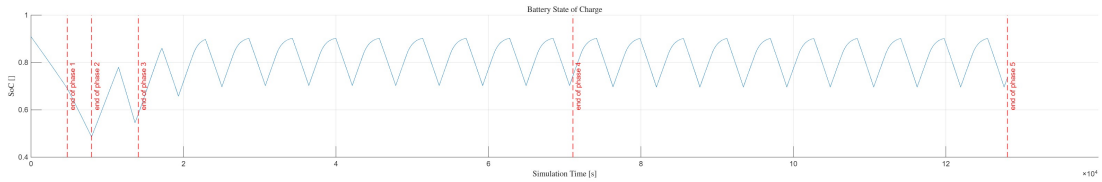

Figure 5.1: State of Charge (SoC) over seconds

This graph shows the State of Charge (SoC) of the battery over time.

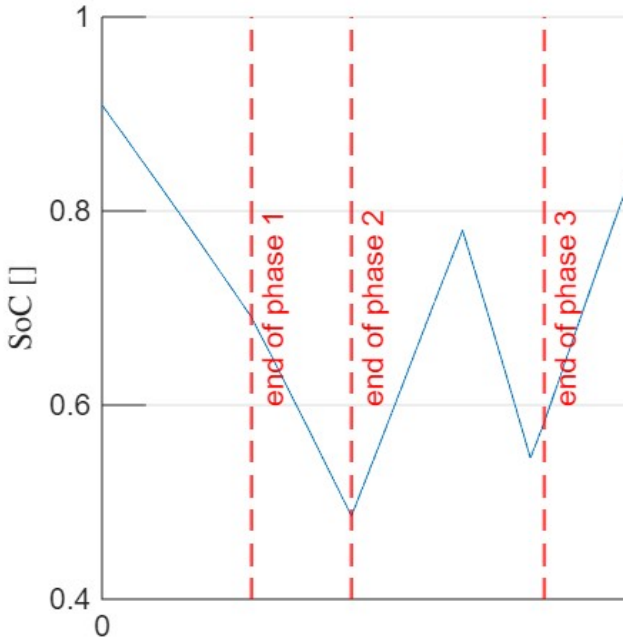

Figure 5.2: First three phases of the battery SoC

During the first three phases the battery is discharging since during launch phase the satellite is not receiving any power from the solar array. During the ground and launch phase and the launcher separation to SA deployment, the power generation is always equal to zero. From "solar panels deploy" onwards, the orbital phase begins, during which power is supplied by the solar panels in the sunlight phase and by the on-board batteries in the eclipse phase. This behaviour is also visible and similar to one shown in the IOS power budget document.

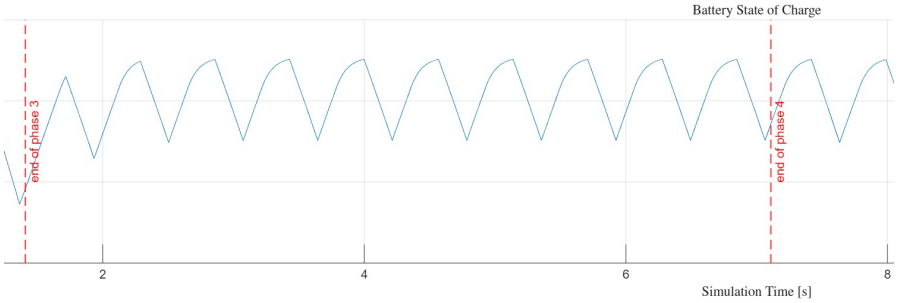

Figure 5.3: Last phase of the battery SoC

During the last phase the battery is constantly switching between charging and discharging. This is due to the fact that during this phase the satellite is transmitting data using both S-band and X-band transmitters. This means that the power consumption is higher than the power generation during the sunlight phase. This behaviour is also visible and similar to one shown in the IOS power budget document. The state of charge does not ever reach the minimum value required by the power budget.

5.2.2 Battery Current over seconds

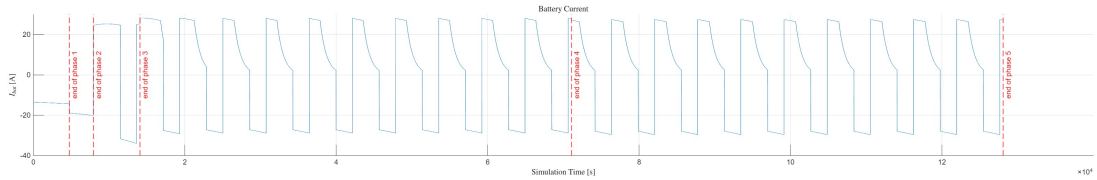

Figure 5.4: Battery Current over seconds

This graph shows the current of the battery over time.

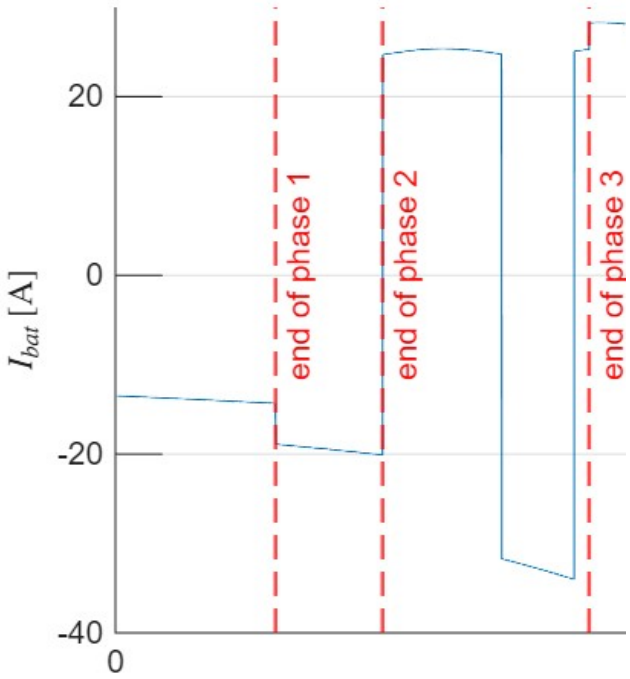

Figure 5.5: First three phases of the battery current

During the first three phases the battery is discharging since during launch phase the satellite is not receiving any power from the solar array. During the ground and launch phase and the launcher separation to SA deployment, the power generation is always equal to zero. From "solar panels deploy" onwards, the orbital phase begins, during which power is supplied by the solar panels in the sunlight phase and by the on-board batteries in the eclipse phase. This behaviour is also visible and similar to one shown in the IOS power budget document.

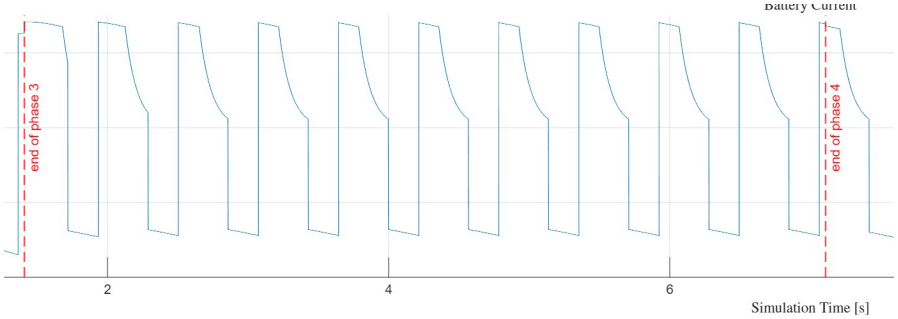

Figure 5.6: Last phase of the battery current

During the last phase the battery is constantly switching between charging and discharging. This means that once the current reaches a certain threshold, the battery management system must quickly respond to prevent overcharging or excessive discharging, ensuring the longevity and reliability of the battery pack. Meaning that the current arriving to the Battery can be constant at times, allowing the battery to never overcharge or discharge too much.

5.2.3 VBus Voltage over seconds

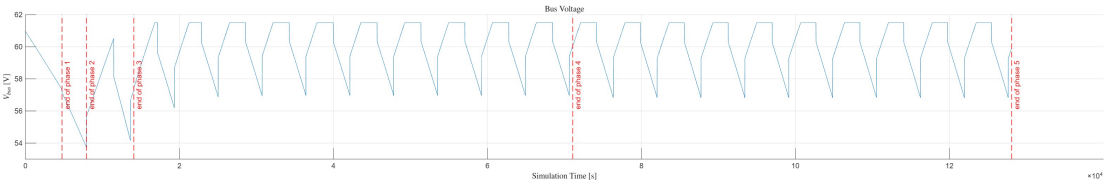

Figure 5.7: VBus Voltage over seconds

This graph shows the Voltage over the Bus between battery and PCDU over time. As mentioned early this is calculated as :

$VBus = V_{battery} + R_{equivalent} \cdot I_{battery}$

Maeaning that depending on the current flowing through the battery, the voltage will be higher or lower than the battery voltage.

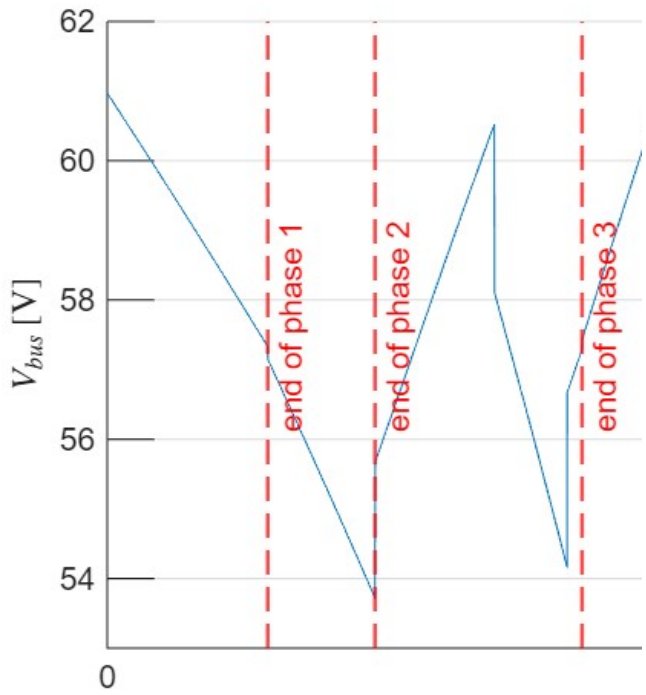

Figure 5.8: First three phases of the VBus voltage

During the first three phases the battery is discharging since during launch phase the satellite is not receiving any power from the solar array so the VBus voltage is decreasing over time.

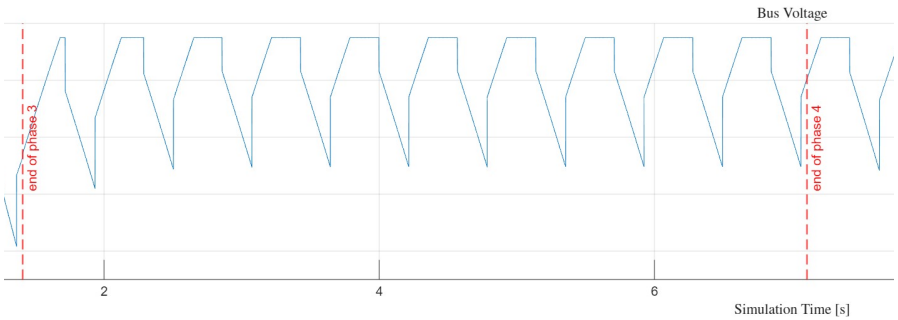

Figure 5.9: Last phase of the VBus voltage

Once the satellite reaches phases 4 and 5 the VBus stabilizes and starts to oscillate between a minimum and maximum value. This is due to the fact that during this phase the satellite is transmitting data using both S-band and X-band transmitters. This means that the power consumption is higher than the power generation during the sunlight phase. This behaviour is also visible and similar to one shown in the IOS power budget document.

5.2.4 MILBUS1553 Interface Testing

The testing of the MILBUS1553 interface is carried out using a dedicated tool called **ECHO**. This software is executed on a separate virtual machine running Windows 10 and provides the capability to generate and transmit telemetry packets directly to the On-Board Computer (OBC) emulator hosted on SIMSAT. The ECHO tool has been designed to support flexible and customizable testing campaigns, enabling the operator to create **Packet Utilization Standard (PUS)** packets that can be precisely tailored to the needs of the mission under study. Once these packets are sent, the code previously discussed within the simulator receives them, analyzes their contents, and automatically forwards the relevant command and data to the MILBUS1553 interface. This process makes it possible to validate not only the implementation of the bus interface itself, but also the correct parsing, handling, and subsequent execution of telecommands at the OBC level. By confirming whether a telecommand has been executed successfully or rejected, engineers can assess the robustness of the entire communication chain, from packet generation to final system response.

In order to start the testing procedure, several preliminary steps must be completed. The first step is to launch the SIMSAT environment, which is accomplished by running the `./SIMSAT.sh` script in the terminal. Once SIMSAT is active, attention shifts to the ECHO virtual machine. Inside this environment, the executable `iride_if.exe` must be run and its configuration verified. Particular care must be taken to ensure that both the `TC_Server` and `TM_Server` share the same IP address, since any mismatch would prevent proper packet routing.

Furthermore, the `SCid` and `Satellite_ID` parameters must match the identifiers assigned to the spacecraft under simulation. This ensures that packets generated by ECHO are correctly interpreted by the OBC emulator as belonging to the simulated satellite. The configuration process is illustrated below:

```
[TCPIP]
Echo_Server.TC_port=9004
Echo_Server.TM_Pkt_port=9003
Echo_Server.TM_Frm_port=7001
Echo_Server.Setup_port=9002
Echo_Server.Message_port=9013
Obe_Server.port=7002
IF_AUX_TC_Server.port=9009
TM_Server.port=3906
TM_Server.address=192.168.136.128
```

Figure 5.10: TCP/IP Configuration

Once the configuration is finalized, the communication between the ECHO tool and the SIMSAT OBC emulator is established through a **TCP/IP connection**. This connection forms the backbone of the testing setup, enabling the seamless injection of telemetry data into the OBC emulator. The virtual machine hosting SIMSAT must be configured with the appropriate IP address, which serves as the target endpoint for the ECHO tool. Once the handshake has been successfully completed, ECHO is able to emulate realistic mission conditions, sending both nominal and erroneous telemetry data for testing and verification purposes. When the tool is running correctly, and the `iride_if.exe` interface is active, the connection is acknowledged as shown in the following figure:

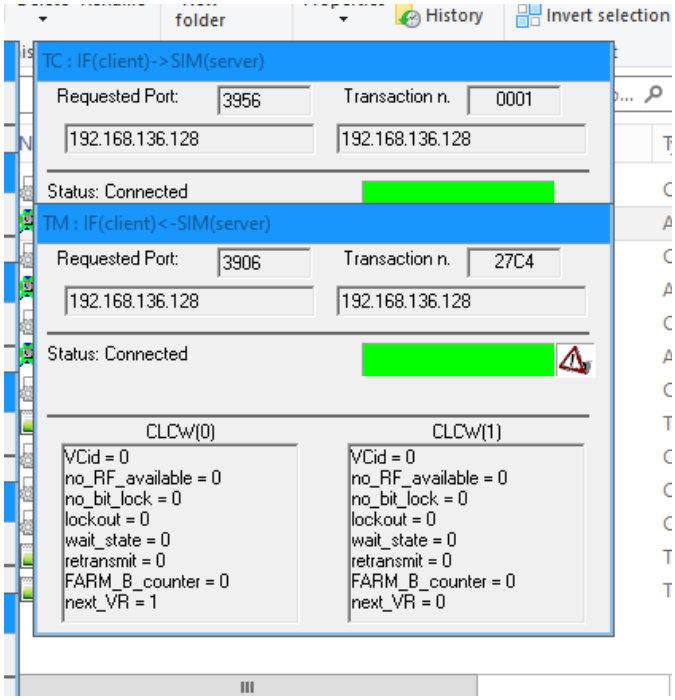

Figure 5.11: Connection accepted by the ECHO tool

The strength of the ECHO tool lies in its user-friendly **Graphical User Interface (GUI)**, which provides operators with the ability to define the specific service type and telemetry packets to be transmitted. The operator can select services and subservices in accordance with the PUS standard and inject telecommands tailored to specific subsystems of the spacecraft. Once the telecommand is sent, the OBC emulator generates a response that allows the operator to verify the success or failure of the command. This iterative exchange enables rapid identification of issues in telecommand parsing or bus communication. Moreover, ECHO generates a detailed communication log, which becomes a powerful debugging tool during development and validation. The logs provide a record of transmitted commands, received acknowledgements, and error reports, which can be analyzed retrospectively to reconstruct the sequence of events and identify root causes of anomalies.

During these interactions, the OBC produces specific **telemetry responses (TM)** that reflect the state of the command processing pipeline. Each response is characterized by a combination of service and

subservice identifiers, making it possible to trace the exact status of the telecommand lifecycle. For instance, the OBC returns TM(1,1) when a telecommand has been correctly received and validated, while TM(1,2) indicates that the telecommand was received but failed validation. In the latter case, additional diagnostic information is included to help developers understand the reasons for rejection. The TM(1,3) response communicates that the telecommand has entered the execution phase, while TM(x,x) represents telemetry generated in response to the executed command and transmitted back to the ground. Finally, TM(1,7) indicates that the execution was successfully completed, confirming that the system has acted on the injected telecommand. This layered feedback mechanism provides assurance that commands traverse the entire processing chain reliably, from reception to execution.

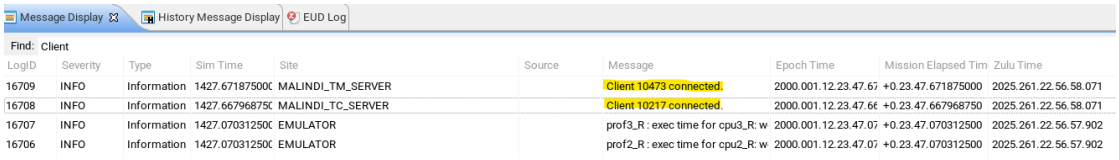

Figure 5.12: SIMSAT connection accepted by the ECHO tool

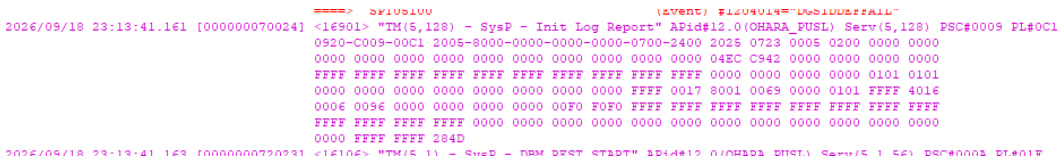

Figure 5.13: ECHO connection configured correctly

The telemetry responses described above form the backbone of the testing and verification methodology. They provide the necessary insight to confirm whether the MILBUS1553 interface and the OBC software behave as expected under a variety of conditions. This methodology allows engineers to monitor the entire communication chain in a controlled and repeatable environment, significantly reducing the risks associated with integration and increasing confidence in the correctness of the implementation. By simulating real-world communication

scenarios and verifying system behavior step by step, the likelihood of mission-critical failures caused by untested telecommand handling is greatly diminished.

The telemetry itself is transmitted in **raw format**, meaning that the PUS packet is sent as a sequence of bytes without any intermediate abstraction. The OBC is responsible for decoding the raw packet, extracting the embedded MILBUS1553 command and its associated data word, and processing them accordingly. This direct approach ensures that the testing environment remains faithful to the actual operational context, where spacecraft receive and process raw telemetry from ground control. The figure below shows an example of a raw PUS packet transmitted by the ECHO tool:

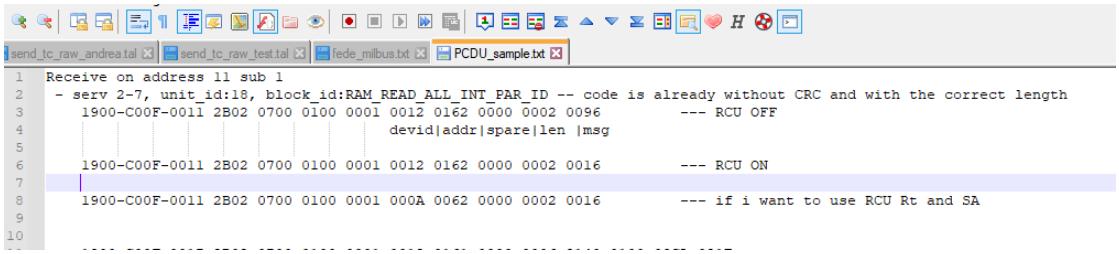

Figure 5.14: Demo PUS Packet sent by the ECHO tool

The hexadecimal representation of the packet highlights the embedded MILBUS1553 command and the associated data word. Each packet corresponds to a distinct telecommand and contains all the necessary information to drive a specific subsystem behavior. By inspecting the comments next to the hexadecimal values, one can clearly identify the role of each packet and its intended effect on the OBC. This level of visibility into the communication process enables developers to confirm that the simulator correctly interprets PUS packets and forwards the proper MILBUS1553 instructions, thereby validating both the protocol implementation and the subsystem integration.

Chapter 6

Conclusion and Future Work

This chapter will discuss the main objectives achieved during the development of the thesis and the prospective future work that can be done to improve the models and the simulator. As outlined in the introductory chapter, the initial objectives were as follows:

- Development and testing of an EPS system model
- Development and testing of a MILBUS1553 interface
- Development and testing of a power line interface
- Compliance with the ECSS standards for the entire simulator

Each of these objectives has been successfully achieved. The developed models have been thoroughly tested and validated to ensure coherence with the mission statements and requirements defined by the IOS mission documentation, as demonstrated in Chapter 5.

6.1 Future Work and Implementation

Looking ahead, there are significant opportunities for further development and enhancement. The full implementation of the entire avionics

system remains an expansive task, demanding extensive work and rigorous testing due to the inherent complexity of the system. Critical components such as the Robot Control Unit (RCU) and the Telemetry, Tracking, and Command (TT&C) system require integration and comprehensive validation to guarantee robust functionality.

The engineering process for the simulator should continue following an iterative methodology, replicating the software engineering core steps employed throughout this thesis:

iterative

- Requirement analysis
- UML design
- Implementation
- Testing and validation

Adhering to this structured approach is imperative to deliver software that is both robust and reliable.

Regarding refinement and further implementation particularly for the EPS and other systems developed during this research, several improvements can be made. Firstly, the battery model design and its source code should be polished and optimized by removing debugging artifacts and extraneous comments, which will enhance maintainability and performance.

The MILBUS1553 interface should evolve to support not only Service 2 of the Packet Utilization Standard (PUS) but also the full spectrum of service types and their subservices. This entails incorporating the Interface Control Document (ICD) of the Power Conditioning and Distribution Unit (PCDU), currently under completion by the avionics team, into the interface logic. Once done, the interface will handle and execute the comprehensive set of telecommands defined for the IOS mission, thereby increasing the fidelity and completeness of command handling.

Similarly, the power line interface requires extensive additional testing and expansion. More instances of power line interfaces should be

developed and integrated corresponding to each terminal and component on the satellite, specifically from the PCDU to various subsystems requiring power. This distributed power modeling will improve the accuracy and realism of power distribution simulations, crucial for mission success.

From the broader perspective of simulator development and based on ECSS standards for simulators and software, there are further enhancements that can be pursued. ECSS-E-ST-40-08 for simulator configuration and ECSS-E-ST-40-07 for software engineering provide guidance on creating modular, reusable, and interoperable simulation models. Leveraging these standards supports the modularization and portability of simulation components, allowing improved reuse across different projects and simulation environments. Following these principles, future work could focus on refining the simulator infrastructure to better support model assemblies, scenario configuration, and integration with external systems, ensuring compliance with emerging space simulation standards.

Additionally, incorporating other avionics subsystems such as the RCU and TT&C in adherence to these standards will promote an integrated simulation platform capable of supporting rigorous mission-level testing and verification. The iterative software engineering process, guided by UML-based design and compliance to ECSS software engineering recommendations, ensures systematic progress towards a resilient and extensible spacecraft simulation model.

In conclusion, the future work on this thesis should focus on expanding subsystem coverage, completing PUS service handling, enhancing power system simulation granularity, and rigorous adherence to applicable ECSS standards for space simulation software. This will ensure that the simulator evolves into a comprehensive, standards-compliant tool for spacecraft avionics development and qualification.

If further detailed elaboration on any particular future work aspect or standards applicability is needed, it can be provided.

6.2 Final Remarks

This thesis has provided a comprehensive overview of the development and implementation of an EPS system and a MILBUS1553 interface within a spacecraft simulator. The work undertaken has successfully met the initial objectives, including the design, implementation, and testing of these components, ensuring their functionality and integration within the larger simulation framework. The adherence to ECSS standards throughout the development process has further strengthened the reliability and robustness of the implemented systems, paving the way for future enhancements and extensions. The iterative approach adopted has facilitated continuous improvement, allowing for the incorporation of feedback and lessons learned throughout the development lifecycle. The successful completion of this thesis has laid a solid foundation for future work in the area of spacecraft simulation and avionics systems. This includes potential enhancements to the EPS system, further integration of the MILBUS1553 interface, and exploration of additional avionics components and their interactions within the simulation environment. The experience and knowledge gained through this project will surely contribute to the advancement of spacecraft simulation technologies and methodologies. The journey undertaken in this thesis has been both challenging and rewarding, and it is hoped that the outcomes will serve as a valuable resource for future researchers and practitioners in the field of aerospace engineering and simulation.

Bibliography

[1] *Applying Secure Software Engineering (SSE) Practices to Critical Space System Infrastructure Development*. May 2016. URL: https://arc.aiaa.org/doi/pdf/10.2514/6.2016-2392 (cit. on pp. 3, 5).

[2] *Space Data Link Security Protocol*. July 2022. URL: https://ccsds.org/Pubs/355x0b2.pdf (cit. on pp. 5–8).

[3] *Packet Utilization Standard*. Sept. 2024. URL: https://ecss.nl/wp-content/uploads/2024/10/ECSS-E-ST-40-08C-DIR1(30September2024).pdf (cit. on p. 23).

[4] *Electrical & ElectronicsBackground reading*. Sept. 2021. URL: https://ecss.nl/wp-content/uploads/2021/09/STC_21-P-Electrical-and-Electronic_background_reading.pdf (cit. on pp. 29, 31).

[5] *Power Line Interface*. Jan. 2015. URL: https://indico.esa.int/event/180/contributions/1353/attachments/1253/1478/1400-Pham_Paper.pdf (cit. on p. 46).

[6] *Interface and Communication Protocol for MIL-STD-1553B Data Bus Onboard Spacecraft*. Apr. 2017. URL: https://ecss.nl/standard/ecss-e-st-50-13c-interface-and-communication-protocol-for-mil-std-1553b-data-bus-onboard-spacecraft/ (cit. on p. 48).

[7] *PUS Telecommand Packet*. Apr. 2011. URL: https://www.researchgate.net/figure/PUS-Telecommand-Packet_fig2_228890657 (cit. on p. 49).