Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

A.y. 2024/2025

Graduation Session October 2025

Simulation of Electrical Power System in a Satellite Digital Twin

Supervisor:

Paolo Maggiore

Tutor:

Ing. Carlo Paccagnini

Candidate:

Giuseppe Bruno

Abstract

The In-orbit Service (IOS) mission is designed to extend the lifespan of spacecraft in orbit by enabling operations such as refueling, maintenance, and corrective interventions. Within this context, the present thesis, conducted in collaboration with Thales Alenia Space, focuses on developing Digital Twin version 1.0.0 for the IOS mission, with particular attention to the Electric Power Subsystem (EPS).

The main objective is to enable testers to safely verify, optimize, and validate spacecraft performance through simulation before launch. Given the company's long-standing collaboration with the European Space Agency (ESA), one of its existing tools, **SIMULUS**, was selected as the simulation framework. Within this framework, the EPS has been modeled in C++ with a focus on the **Power Control and Distribution Unit (PCDU)** and its associated power lines, which are responsible for managing and distributing electrical power to the spacecraft's subsystems. The developed module has been integrated into SIMULUS to create a reliable and extensible simulator. Furthermore, the interface between the EPS and the On-Board Computer (OBC) has been simulated via the MIL-STD-1553B data bus and the Packet Utilization Standard (PUS) protocol, enabling telemetry exchange with ground stations.

The simulator enables evaluation of the PCDU's behavior, assessment of different operational scenarios, and analysis of edge cases that would be difficult to reproduce experimentally. Although the scope of the thesis is limited to software simulation, the work establishes a foundation for future improvements and practical applications in satellite mission development.

Table of Contents

List of Figures IV

1 Introduction 1
- 1.1 Satellites: Applications, Challenges, and the Role of Power Management . . . 1
- 1.2 The Role of Simulation in Space Engineering . . . 2
- 1.3 In Orbit Servincing Mission . . . 3
- 1.4 Thesis Scope, and Objectives . . . 4
- 1.5 Thesis structure . . . 5

2 State of the Art 6
- 2.1 The C++ Programming Language . . . 6
- 2.2 Model-Based Design with MagicDraw and UML . . . 7
- 2.3 European Cooperation for Space Standardization Simulation Model Portability Standard ECSS - SMP . . . 8
- 2.4 SIMULUS . . . 8
- 2.5 Avionic Communication Protocols . . . 9
  - 2.5.1 Packet Utilization Standard . . . 9
  - 2.5.2 Milbus . . . 9
- 2.6 Rationale for Technology Selection . . . 10

3 Simulator Architecture 12
- 3.1 Spacecraft Machine . . . 13
  - 3.1.1 TEMU Emulation Utility . . . 14
  - 3.1.2 Software Infrastructure for Modelling Satellites (SIMSAT) . . . 15
- 3.2 Ground Station Machine . . . 16

4 Implementation and Development Process 19
- 4.1 Power Control and Distribution Unit overview . . . 19
  - 4.1.1 Serial Shunt Switching Regulators . . . 21
  - 4.1.2 Regulated and Unregulated bus . . . 21

4.1.3 Battery Charging Module (BCM) . . . 22
4.1.4 Latching Current Limiter . . . 22
4.1.5 Heater Drivers . . . 22
4.2 UML implementation . . . 23
4.3 ECSS-SMP compliancy . . . 27
4.4 Pcdu Main Object Implementation . . . 28
4.4.1 PulseRaisingPort . . . 30
4.4.2 Battery Charge Module Implementation . . . 31
4.4.3 Unregulated Bus Implementation . . . 32
4.5 Milbus Development . . . 34
4.5.1 Milbus Overview . . . 34
4.5.2 Milbus Implementation . . . 35
4.6 Pcdu Outgoing Interface Implementation . . . 37
4.7 Powerline Implementation . . . 38
4.8 Battery and Solar Array Interface . . . 40

5 Validation and Verification 41
5.1 From Launch to In-Orbit Operations: Mission Phases . . . 42
5.2 Rendezvous Phases . . . 44
5.3 Ground Segment Testing . . . 45
5.4 PUS overview . . . 48

6 Conclusion 50

Bibliography 52

List of Figures

1.1 IOS satellite 4

2.1 PCDU UML 7

3.1 Environment Configuration 13
3.2 Emulation/Simulation Boundary Diagram 14
3.3 MMI[7] 15

4.1 PCDU schema 20
4.2 PCDU UML 24
4.3 PCDU tree hierarchy 25
4.4 PCDU XML 26
4.5 Main XML file 27
4.6 Battery charging profile 31
4.7 milbus architecture 35
4.8 Command word bit usage 35
4.9 powerline architecture 39

5.1 Loads power, SA power, sun/eclipse flag 43
5.2 SOC, battery current and bus voltage 43
5.3 Loads power, SA power, sun/eclipse flag 44
5.4 SOC, battery current and bus voltage 45
5.5 Overview of simulator and ground station connection views. Top row shows the ground and simulator interface views, bottom row shows internal system displays. 47
5.6 TCC after command sent 48
5.7 Telecommand structure 48
5.8 Data field header 49
5.9 PUS string 49

Chapter 1

Introduction

1.1 Satellites: Applications, Challenges, and the Role of Power Management

Satellites are among the most complex systems that the human mind has been able to built. They are used for several purposes, including global navigation and positioning via Galileo or GPS, for making internet accessible to the most remote zones on the Earth with constellations like Starlink, to enable space exploration missions such as Rosetta and supporting weather forecasting. Satellites are built for civilians, scientists and today they have also a strategic role in defense and security. The number of them have grown rapidly in recent years, also for the rise of private companies. The satellite tracking website "Orbiting Now" lists around 13.000 active satellites in september 2025 and according to the analysts from McKinsey & Company[1] the number will be around 27.000

The hostile environment in which they operate makes their development a challenge. Space is characterized by vacuum, radiaton, extreme temperatures variations and limited access to energy. Once deployed in orbit, repairs are rarely possible, so all the loads must be reliable for its lifetime. The loss of just one of the subsystem of the satellite could cause the failure of the entire mission, leading significant finalcial and scientific loss. For this reason the construction of a spacecraft requires rigorus engineering methods and procedures, every component must be follow some standars and tested before launch.

The architecture of a satellites includes several sub-systems, each dedicated for a particular function. These are the communication subsystem, the attitude and orbit control subsystem, the power subsystem and all the specific subsystems for the mission goals. Although each subsystem performs a specific role, they are all interconnected and a failure in one of them could spread into failures in others.

Among them, one particular components is the power subsystem because it ensures that all the others remain operational. Without a stable and sufficient power supply no hardware can be turn on. Making an example, it represents the foundations of a house under construction.

1.2 The Role of Simulation in Space Engineering

Given the high cost and risk of space missions, simulators provide a safe environment for engineers and researchers to test their solutions. They enable analysis, assessment and verification of spacecraft and mission performance ensuring the safety, efficiency and the success of complex commands before they are executed in the real environment [2]. This possibility does not only reduce the risk associated to unwanted behavior of the system but also accelerates development by allowing engineers to quickly test different design scenarios.

Simulation is not used only during the development phase but acroos the lifecyle of the space misison. During the design phase, it helps to evaluate trade-offs between different technologies or architectures. During integration, it ensure that each component functions correctly as expected. Even during operations, they can be used for predict satellites behaviors under specific enviromental conditions. A simulator is able to represents all those scenarios that are difficult to reproduce experimentally like battery depletion or unusual load demand that may damage the spacecraft. If engineers do not want to use software solution for these cases, they would need costly test facilites risking to damage the hardware. As space systems become more complex and more expensive during the years, the reliance on accurante and flexible simulation frameworks is expected to grow.

ESA and Nasa are collaborating to create standardize frameworks to facilitate interoperability across simultation tools used in different missions. This process aim to reduce the costs against the development of simulator tools and simulated models for specific missions. By adopting common standards, these agencies aim to reuse simulation components across different projects, enanching efficiency in misison planning [3].

Future generations of spacecrafts will require lighter mass while being sujbect to higher load and extreme conditions over longer time than the current spacecrafts. Since extreme thermal condition, mechanical and degradation of the components due to radiation may be impossible to reproduce in a laboratory the use of computational simulation assumes a central role. Vehicles will encounter conditions that can not be predicted so a new approach to verification and validation of models, systems

and simulation must be developed. In addition the ability to modify real time parameters for the mission goal will be indispensable.

The new challenges that liea ahead need can not be based just on empirical-based standard but require new multidisciplinary physics-based methods to ensure robustness, reliability and sustainability. If the physics models can be integrated with the on-board sensors they will be able to certificate through simulation supports real-time health management during missions, making the bases of a Digital Twin.

A Digital Twin is an integrated multiphysics, multiscale and probabilistic simulation of a spacecraft that uses physical models, sensors, fleet history to mirror the life of the corresponding flying twin. It is ultra realistic and may include one or more sub-systems of the specified vehicle. By combining all these information, the Digital Twin can monitor system health, the remaining life and assess mission success probability. It can also predict safety-critial events comparing expected behavior and history behaviors. If the Digital Twin is aware by damagaes or degradation it may reccomend changes in mission profile to increase the probability of mission success[4].

1.3 In Orbit Servincing Mission

The IOS mission is designed to extend the lifespan of spacecraft in orbit by enabling operations such as refueling, maintenance, and corrective interventions. This emerging paradigm encompasses essential operations like inspection, component repair, and transfer of space assets to disposal orbits, representing a significant technological leap.

Italy has strategically positioned itself within this domain through the **National In-Orbit Servicing Demonstration Mission**. Funded by the **National Recovery and Resilience Plan (PNRR)** and managed by the **Italian Space Agency (ASI)**, the project is dedicated to the development and qualification of key enabling technologies for future-generation in-orbit services.

The mission was awarded to a **Temporary Grouping of Companies (RTI)** led by **Thales Alenia Space** as the prime contractor, including major industrial players such as **Leonardo**, **Telespazio**, **Avio**, and **D-Orbit**. The mission design involves a dual-satellite configuration—a **servicer** (an autonomous robotic vehicle) and a **target** satellite. Set for launch by 2026, the demonstration will validate critical capabilities in robotic control, artificial intelligence, and sensing necessary for complex rendezvous and proximity operations.

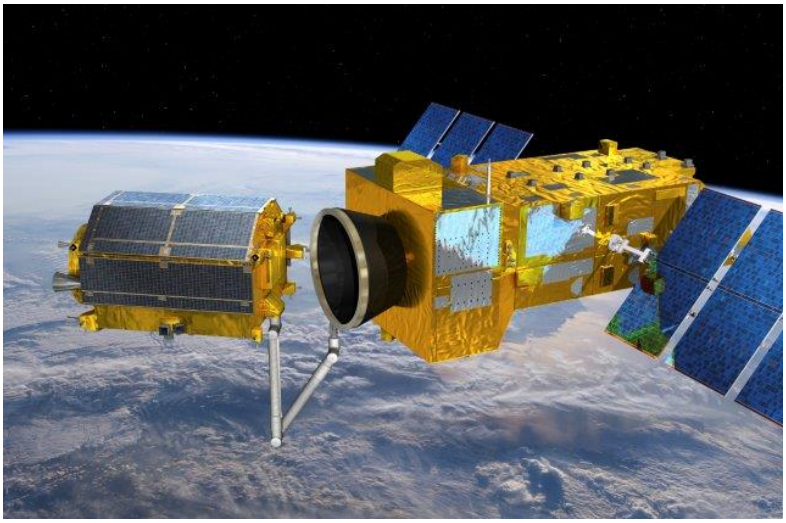


Figure 1.1: IOS satellite

1.4 Thesis Scope, and Objectives

This thesis establishes the foundational work for a **Digital Twin, designated version 1.0.0,** dedicated to the Electric Power Subsystem (EPS) of the Italian In-Orbit Servicing (IOS) mission. This development represents a high-fidelity simulation effort aimed at creating a robust validation environment for spacecraft power management and operational procedures.

The entire simulator environment is realized through **SIMULUS**, which operates on virtual machines and interacts directly with the mission's Ground Segment infrastructure. Due to the critical nature of the application, adherence to the **ECSS-SMP (Spacecraft Model Portability)** standard is rigorously enforced, necessitating that all implementation be performed in `C++`. The architectural basis was first defined using **UML** within the **MagicDraw** environment, establishing a scalable architecture engineered for straightforward maintenance and future feature enhancement.

The central contribution of this work lies in the detailed modeling of the **Power Control and Distribution Unit (PCDU)**, the core component of the EPS responsible for managing and distributing electrical power to spacecraft subsystems. The simulation reproduces its behavior in controlling the activation and deactivation of power lines, regulating power flow toward onboard consumers, and ensuring consistent and safe operation across different mission phases.

- **Power Control and Distribution Unit (PCDU):** The primary focus of this thesis. The PCDU functions as the spacecraft's central energy management

unit, responsible for allocating and supervising power delivered to all electrical loads through a set of controlled power lines.

- **Power Lines:** The physical and logical interfaces that distribute electrical power to the various subsystems of the spacecraft, managed and monitored by the PCDU.

The critical function of the **PCDU** requires it to interface with the simulated On-Board Computer (OBC) through the internal **MIL-STD-1553 (Milbus)** protocol, receiving commands for load management and status updates.

A crucial component of this work was enabling seamless communication between the Ground Station and the simulated spacecraft. This was achieved by implementing the external telemetry and telecommand exchange using the **Packet Utilization Standard (PUS)**. Consequently, this thesis delivers a functional, standard-compliant EPS model in `C++`, establishing a solid foundation for the IOS mission's comprehensive Digital Twin and future extensions to the complete power subsystem.

1.5 Thesis structure

The thesis is structured into the following chapters. Chapter 2 explains the state of the art of the software and programming languages used during this thesis. Chapter 3 discusses the simulator architecture used for the simulation with particular emphasis on the SIMULUS framework and on the standard used for programming the models. Chapter 4 details the design and the implementation of the EPS components as well as their integration with the the On-Baord Computer via MIL-STD-1553B and the Packet Utilization Standard (PUS). Chapter 5 is about the results of the simulation, showing how the Digital Twin validates tests on different operational scenarios. At the end, Chapter 6 contains the closing remarks about the thesis with outlining the future implementation that can be added.

Chapter 2

State of the Art

This chapter presents an overview of the technologies, tools and methodologies that has been used for this thesis work. It provides context on the software frameworks, modelling environment and the standards adopted during the development of the simulator. The first section introduces the C++ programming language, outlining its revelance for developing high-performance software for simulation environments. The integration between MagicDraw and UML is then discussed. Subsequently, the SIMULUS framework is presented, describing its role in space simulation and the compatibility with the ECSS standard. Lastly, this chapter discusses the virtual machine environment used to host the development. The rationae behind these technological choices, with a comparison to alternative solutions.

2.1 The C++ Programming Language

C++ is a compiled, object oriented programming language that allows the developer to have fine-grained control over hardware resources and memory. It combines the efficiency and flexibility of low-level programming with the abstraction capabilities of a high-level language, enabling both performance optimization and structured software design. It is statically typed, meaning that the type of every variable is declared or determined at compile time rather than at runtime. It is also a case-sensitive language, distinguishing between uppercase and lowercase identifiers. Since C++ source code is compiled directly into machine-specific instructions, programs typically run faster and more efficiently than those written in interpreted languages, where code is executed line by line. Furthermore, C++ is officially supported by the ECSS standard, as well as by development tools such as MagicDraw and SIMULUS, making it a suitable choice for software engineering in the space domain.

2.2 Model-Based Design with MagicDraw and UML

The Unified Modeling Language (UML) is a general-purpose, object-oriented graphical language used to represent the architecture and design of a system. It provides a standardized way to describe the behavior, interactions, and structure of the various components that compose a system.

In this thesis, UML has been employed to design the Power Control and Distribution Unit (PCDU) and to model its interactions with other components within the simulator. As UML is a standardized language that requires specialized software for its interpretation, MagicDraw has been used for this purpose.

MagicDraw, developed by Dassault Systèmes, has been widely adopted in the aerospace and systems engineering domains since its early development. From its inception, it has been designed with a strong focus on model-based system engineering (MBSE), making it particularly suitable for complex aerospace applications. Furthermore, MagicDraw is officially supported by the SIMULUS framework and enables the automatic generation of code compliant with the adopted standards through UML model import.

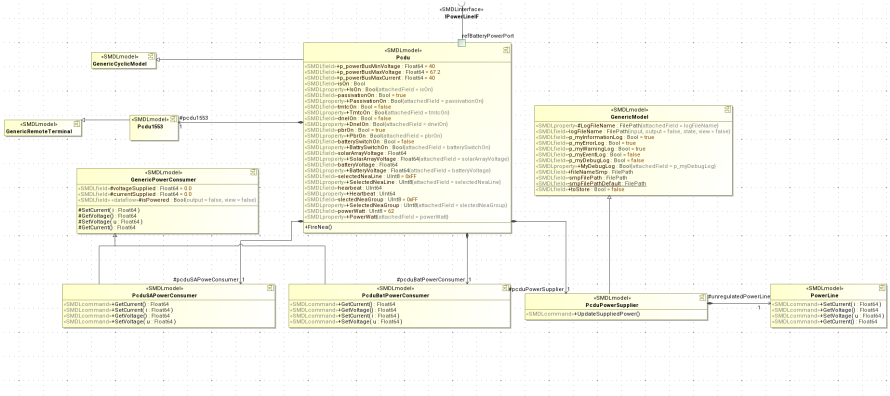


Figure 2.1: PCDU UML

The image above represents the PCDU UML that will be discussed in the next chapters.

2.3 European Cooperation for Space Standardization Simulation Model Portability Standard ECSS - SMP

The ECSS-SMP standard is a set of guidelines for simulation models developed by the European Space Agency (ESA) in collaboration with other organizations in the European space industry, such as the European Space Operations Centre (ESOC). It is widely adopted for the engineering of simulation software for space systems.

The standard defines the structure and behavior of simulation environments and models, providing all the necessary specifications and tools for building a space simulator that is compliant with ECSS requirements. One of its main advantages is the ability to reuse simulation models across different projects and simulators, as long as each simulator adheres to the SMP standard.

Each simulation model is designed as a stand-alone component. This modularity allows users to simulate complex systems, such as satellites, by selecting and connecting the required models according to the standard and assigning appropriate values to ensure compliance with the specified requirements.

2.4 SIMULUS

SIMULUS is a modular framework for the development of spacecraft simulators, developed by the **European Space Agency - European Space Operations Centre (ESA/ESOC)** and based on the ECSS-SMP standard [5]. It provides tools for building high-fidelity mission simulations, including CPU emulators (ERC32, LEON2, LEON3) and generic models representing spacecraft subsystems. The framework is designed to be flexible and scalable, allowing developers to select components according to mission requirements and to create models tailored for specific scenarios. By adhering to ECSS-SMP, SIMULUS ensures model interoperability, reusability, and traceability across different simulation projects.

A core part of SIMULUS is **SIMSAT**, the software infrastructure for modeling satellites. SIMSAT offers a GUI-based Man-Machine Interface (MMI) to monitor and control simulations in real time. Through SIMSAT, users can observe model parameters, execute commands or scripts, log simulation events, and save or restore states. This environment supports interactive testing, scenario exploration, and debugging, making it suitable for both development and operational verification of satellite systems.

SIMULUS employs a component-based approach where models represent spacecraft subsystems or functional units, and services provide global simulation functionalities. Models are uniquely identified via UUIDs, configurable for mission-specific parameters, and can be dynamically connected through well-defined interfaces.

This design enables modularity, facilitates integration with UML/MagicDraw, and supports a wide range of simulation types, from functional and analytical models to hardware emulators like the PCDU, battery, or solar array.

2.5 Avionic Communication Protocols

During the thesis development, two distinct **avionics communication protocols** were utilized: the **Packet Utilization Standard (PUS)** and **Milbus**. The **PUS** is employed for **space-to-ground and ground-to-space telemetry exchange**, handling the sending and receiving of vital spacecraft data. In contrast, **Milbus** serves as the **internal satellite data bus**, facilitating information exchange between various onboard subsystems and units. This internal function is broadly analogous to the role of **Ethernet** in a standard computer network, providing a local communication link between connected components.

2.5.1 Packet Utilization Standard

The **Packet Utilization Standard (PUS)**, defined under the **European Co-operation for Space Standardization (ECSS-E-ST-70-41)**, is the core protocol for **space-to-ground and ground-to-space communication**. It governs the structure and content of all data exchanged between the spacecraft and the ground control segment. PUS establishes a standardized framework for **Telemetry (TM)** and **Telecommand (TC)** services, categorizing them into functional domains (e.g., On-Board Operations, Data Management, Housekeeping) via predefined **Service Types** and **Subtypes**. This modular, service-oriented structure is crucial for defining, managing, and verifying the execution of operations, facilitating robust and interoperable command and control across various missions.

2.5.2 Milbus

The **MIL-STD-1553** is a widely adopted military standard that defines the mechanical, electrical, and functional characteristics of a time-division multiplexed (TDM) serial data bus. While initially designed for military avionics, its high reliability and robust architecture led to its widespread use in **spacecraft on-board data handling (OBDH)** subsystems for both military and civil applications. The protocol is structured around a central **Bus Controller (BC)** that governs all communication with connected **Remote Terminals (RTs)**, ensuring strict control over data flow. Furthermore, it is characterized by a half-duplex **command/response protocol** and commonly features **dual-redundant balanced line physical layers** to maximize fault tolerance.

2.6 Rationale for Technology Selection

The selection of both the programming language and the simulation environment was guided by the need for high computational efficiency, real-time performance, and direct control over hardware interactions—requirements paramount for an accurate satellite simulator.

While numerous languages are available for software development, including C, Python, and Java, C++ was selected as the optimal choice for its unique balance of performance and object-oriented simplicity.

- **C:** As a low-level language, C offers absolute control but necessitates extensive manual implementation of fundamental features, such as object-oriented concepts and complex library integrations. This requirement for excessive low-level programming compromises development speed and code maintainability, despite offering high performance.

- **Python:** Python is a high-level, interpreted language. Although it excels in rapid prototyping and data analysis, its interpreted nature leads to inefficient execution unsuitable for the strict, real-time requirements of a performance-critical simulator kernel.

- **Java:** Java, which relies on the Java Virtual Machine (JVM), sacrifices direct memory management for portability. This intermediate layer and garbage collection mechanism introduce overhead and remove the critical low-level control necessary to guarantee the predictable performance required for time-sensitive avionics applications.

- **Rust:** While languages like Rust offer excellent speed and memory safety, its relative novelty means that many established avionics frameworks, supporting tools, and legacy systems—including those required for this project—lack mature support, making its adoption impractical at this time.

In conclusion, **C++** was chosen as it provides native support for high-performance Object-Oriented Programming (OOP), direct memory management, and the compilation to native machine code, satisfying the strict efficiency and control requirements of the simulator.

Alternative simulation tools were evaluated against the primary objective: creating a high-fidelity satellite subsystem simulator.

- **MATLAB and Simulink:** These commercial tools from MathWorks are highly effective for specific tasks. **MATLAB** is designed primarily for numerical algorithms and data analysis, while **Simulink** excels at graphical

Model-Based Design for simulating dynamic systems (e.g., control laws, electrical models). However, they lack the specific architecture and interfaces designed for integrating with a space mission control system and are not optimized for full, high-speed, operational spacecraft simulation.

- **SIMULUS:** Developed specifically by the European Space Agency (ESA), SIMULUS is an operational simulation framework built explicitly for spacecraft simulation. Its key advantages are its inherent ability to model all satellite subsystems, its native consideration of mission time, and its perfect integration with C++ (the language of choice), providing a robust platform tailored precisely to the project's objectives.

Chapter 3

Simulator Architecture

This chapter provides a detailed examination of the simulator's architecture, including the underlying physical hardware and the rationale guiding the chosen virtualization strategy.

The simulator resides within a virtualized environment hosted on a single, high-specification physical workstation. The system is built around an **Intel Core Ultra 9 285K processor**, supported by an ample **128 GB of RAM**, and utilizing a **2 TB Solid State Drive (SSD)** running the **Windows 11** operating system. This robust hardware foundation was indispensable; the computational load imposed by the high-fidelity **SIMULUS** framework, particularly when modeling complex subsystems, necessitated a strong resource baseline to guarantee efficient and reliable operation.

The entire platform is implemented using **VMware Workstation Pro 17**, adopting a strategy of **strong virtualization**. This approach was intentionally chosen over light virtualization, predominantly for two critical reasons: the enhanced **isolation and security** it provides, protecting the host machine's filesystem from any potential compromise within the simulation environment, and its ability to grant the guest operating systems greater independence from the inherent virtualization challenges of the Windows 11 host.

The available physical resources were meticulously allocated across the two primary virtual machines based on their functional demands:

- **Spacecraft Simulator Machine:** As the kernel for the performance-intensive **SIMULUS** framework, this VM received the majority of the resources, specifically allocated **16 virtual cores** and approximately **100 GB of dedicated RAM**. This high provision ensures the necessary smoothly execution for simulator.
- **Ground Station Machine:** Given that its sole function is constrained to external packet-level communication (sending and receiving PUS data), the

Ground Station VM required fewer resources. Consequently, it was allocated **8 virtual cores** and **10 GB of RAM**.

This deliberate distribution guarantees that critical simulation components receive the essential computing capacity to operate effectively and reliably within the established architecture. Communication between the two virtual machines is facilitated over a dedicated **Local Area Network (LAN)** segment using the **TCP/IP protocol**. This setup effectively isolates the simulation network, enhancing fidelity by ensuring only the Ground Station and the simulated Spacecraft interact. The **Spacecraft machine** hosts a continuous process that operates as the server, listening for incoming connections on a specified network port. Conversely, the Ground Station machine functions as the client, initiating the TCP/IP connection to establish the communication link. To streamline the setup and ensure consistent operational startup—thereby eliminating the need to modify configuration files prior to each simulation run—a **static IP address** was assigned to the Spacecraft server. This practice is crucial for maintaining a reliable, persistent network configuration throughout the development and testing lifecycle.

The image below represents the environment configuration.

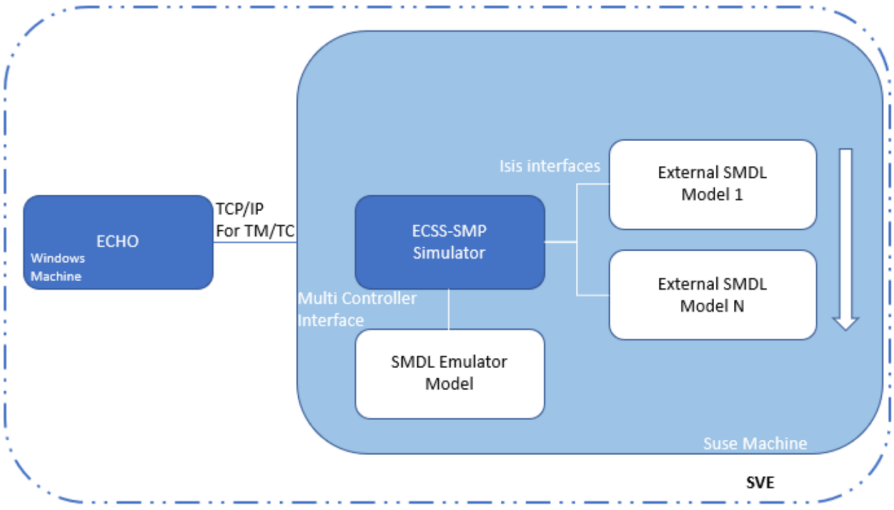


Figure 3.1: Environment Configuration

3.1 Spacecraft Machine

The spacecraft machine runs **SUSE Linux Enterprise Server (SLES)** distribution and acts as the **spacecraft**, where the simulation is executed.

SLES was chosen because it provides a stable, secure and long-term supported environment. Unlike the other Linux distros, this one is ideal for mission-critical

workloads, which aligns with the aerospace industry. Its a long-term support system, ensuring continuity during long development and testing phase without disruptive upgrades. Furthermore, this operating system is optimized for running computationally intensive simulations, such as the heavy numerical processing needed for a digital twin. It also integrates advanced security mechanisms, including confidential computing to protect data during processing and granular user and system permissions protecting an environment full of sensitive information [6].

Within the spacecraft VM, satellite is simulated using **SIMULUS**. Before proceeding, its important to clarify the distinction between simulation and emulation. The spacecraft VM hosts the On-Board Software (OBSW), which requires the On-Board Computer (OBC) to operate. Since the OBC hardware is expensive and not available in the laboratory, it is **emulated** within SIMULUS using the resources of the virtual machine and **TEMU** Emulator. At the same time, the OBC needs to communicate with the spacecraft loads that are not physically present and that are **simulated** by C++ modules integrated into SIMULUS.

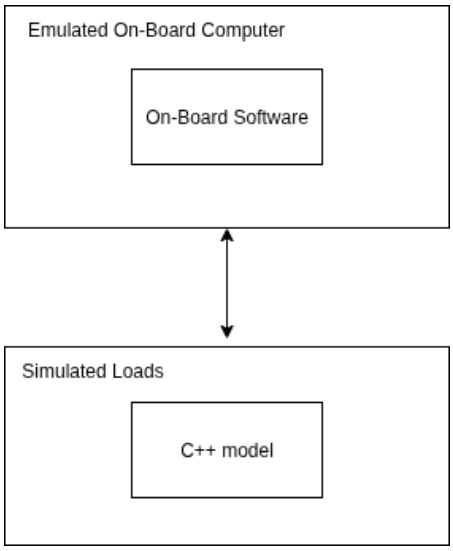


Figure 3.2: Emulation/Simulation Boundary Diagram

3.1.1 TEMU Emulation Utility

TEMU (Terma Mission Utility) is a high-performance commercial tool specifically designed for the **emulation** of spacecraft processor cores and subsystems, often utilized in Software-in-the-Loop (SIL) environments. TEMU utilizes advanced techniques such as dynamic binary translation to achieve high performance

in emulating processor cores from major aerospace families, including SPARCv8 (e.g., LEON3, LEON4, GR740) and PowerPC. Crucially for this project's context, TEMU provides detailed models for standard spacecraft buses and devices, including support for the **MIL-STD-1553** data bus. Its architecture, featuring an easy-to-use device modeling API in C or C++, facilitates seamless integration into existing simulation frameworks and test benches, supporting the development and validation of complex On-Board Software (OBSW) prior to deployment on flight hardware.

3.1.2 Software Infrastructure for Modelling Satellites (SIMSAT)

It is the **SIMULUS Graphical User Interface (GUI)** called as Man Machine Interface (MMI) since it allows to actively monitor and controll the simulation. MMI can display the status of each simulated model such as parameters, services, variables and connections. This software allows the user to control the ongoing simulation via commands and scripts, recording the logs, raising exception in case of anomalous behaviour and saving or restoring the states of the modules. The image below shows the SIMULUS GUI. It provides several features, listed below:

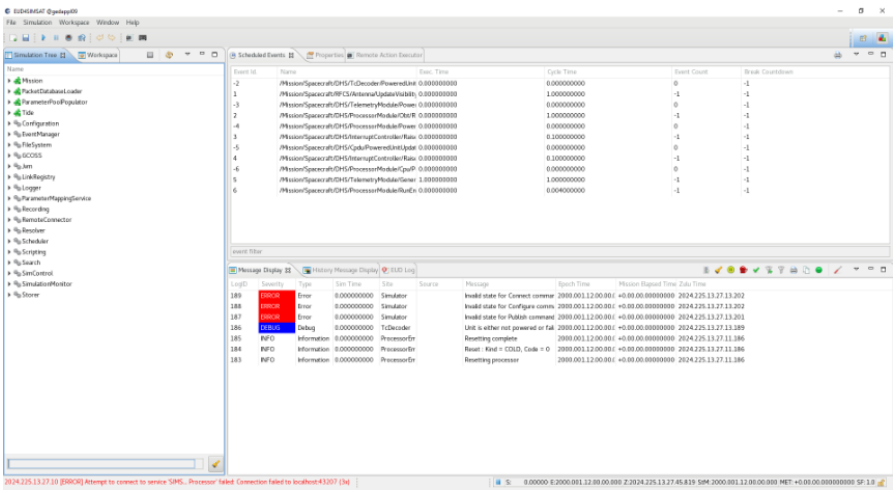


Figure 3.3: MMI[7]

- **Simulation Tree View**: It shows all Services, Models and published Fields/Properties of the attached simulation
- **Message Displays**: Shows current and past messages with all the error information from the MMI itself

- **Events Display**: Overview of all scheduled events and their properties
- **Properties Display**: Shows the properties of the currently selected node in the Simula
- **MAT Displays**: Configurable Table View to track the value of one or more Simulation Parameters/Properties
- **GRD Displays**: Configurable Graphical Display to track the value of one or more Simulation Parameters/Properties
- **Action Executor**: Allows to send commands to th connected script host to interact with the simulation

The Figure 3.3 represents the GUI as the user sees it. On the bottom right there is the Simulation Status bar, it gives information about the **simulation time (S)** in seconds, **Epoch (E)**, **Zulu (Z)**, **Start Mission Time (StM)** and **Mission Elapsed Time (MET)**. It has also a **Speed Factor (S)** allowing the simulation to run faster than the actual time. On top is present the simulation control bar that gives access to severals tools such as saving and restoring the simulation state, run and pause the simulation, change the scheduler configuration. On the left there is the simulation tree. This view allows the user to view and explore the component hierarchy of the attached simulation. Exist multiple kind of nodes, some of them are complex nodes, having child nodes, others can be object or even more granular as variables. Since projects are really complex, so the hierarchy contains a huge number of nodes, a search function is available.

Is important to notice that the connection between the Ground Station Machine and the spacecraft one can not be established unless SIMSAT is running, since it has a process listening on a port.

Inside a folder on the desktop named **tasi.sve** are presented different folders, **Src** with all the satellite simulated model in C++, **Simulator** containing the scripts in order to run SIMSAT and **target** containing all the scripts and the configuration files the setup the simulator and the new models. This structure is revelant since in the implementation part this terms will return.

3.2 Ground Station Machine

This virtual machine, named **Echo**, which operates on the Windows 7 operating system. This specific OS was chosen due to its essential compatibility with the proprietary internal frameworks utilized for ground segment operations. The machine's primary function is to host the operational software necessary to establish, monitor, and control the communication link with the simulated spacecraft.

Establishing and maintaining communication with the spacecraft simulator requires sequential execution of several interconnected software applications. These tools are critical for translating user commands into network packets and interpreting incoming telemetry.

- **iride_if.exe (Interface Manager):** This software is responsible for establishing the fundamental **TCP/IP connection** between the **Echo** and **Spacecraft** VMs. During initialization, it loads a configuration file containing all necessary network parameters, including static IP addresses and dedicated communication ports. The use of static IP addresses is critical here, as it ensures consistent and repeatable connection establishment without requiring configuration file modification for every simulation run. Furthermore, the configuration specifies distinct ports for Telecommand (TC) transmission and Telemetry (TM) reception, a requirement dictated by the SIMSAT framework.

- **Message Transport Protocol (MTP):** The MTP software defines the necessary rules and mechanisms for reliably transmitting messages across the network nodes. Its function is to ensure that data is delivered correctly, in sequence, and without duplication or loss, handling low-level tasks such as error detection and flow control. The integrity of the ground system is verified during the MTP boot phase, as it checks all configuration files; any mismatch in network parameters (e.g., altered IP or port settings) indicates a configuration fault.

- **OBE4.exe (Packet Decommutation and Monitoring):** OBE is a crucial monitoring tool that reads the outgoing Telecommands (TCs) and the incoming Telemetries (TMs). Its core function is to **deserialize** the received data packets, making them understandable to the user via a Graphical User Interface (GUI). This process relies on a pre-loaded dataset of known commands for proper interpretation; unknown packets will be flagged (typically with a '?') but will not compromise software stability. OBE allows the user to verify the successful execution of commands, as a corresponding telemetry packet is expected upon every successful TC transmission.

- **Telecommand Console (TCC):** TCC provides the graphical interface that allows the operator to send Telecommands to the spacecraft. While it typically relies on a database of predefined commands, the need to constantly update this database for developmental testing presents a logistical challenge. To bypass this, especially in the context of this thesis where official Telecommands were not fully established, the software permits the transmission of **RAW byte data** directly to the onboard software. Upon execution, TCC sends initial boot Telecommands to the spacecraft; their absence in the OBE monitor

signals either a configuration issue or a failure of the On-Board Software (OBSW) to recognize the initial commands.

This machine provides also all relevant technical documentation and operational manuals, providing immediate support for troubleshooting and maintenance activities.

Chapter 4

Implementation and Development Process

This chapter provides a detailed explanation of the development and validation processes for the **Electric Power Subsystem (EPS)** software model. The primary focus is placed upon the **Power Control and Distribution Unit (PCDU)** and the associated modeling of the satellite's power lines. The discussion includes all simulation techniques utilized, along with a comprehensive justification for the implementational choices adopted. It is important to note that the PCDU developed in this work does not follow the exact requirements of the units used in the IOS missions, due to confidential reason. Instead, a representative and functionally equivalent model has been designed to demonstrate realistic behavior within the simulator. Before proceeding to the in-depth analysis of the software implementation, it is essential to first establish a functional and physical description of the PCDU to properly contextualize the objectives of this modeling.

4.1 Power Control and Distribution Unit overview

The role of this component is to distribute the power generated by the battery and the solar array across the system. It sits between these sources and all the spacecraft loads and unlike other spacecraft's component it operates more independently from the On-Board-Computer (OBC). The unit features two main interfaces, one for each power source, the electronic to handle the power distribution, the dedicated powerline interfaces for the loads and also a module to exchange Milbus messages with the On-Board-Computer, acting as remote terminal.

In addition to its distribution role, the PCDU protects power lines to prevent failure propagation between loads and the electrical power subsystem (EPS), provides load switching capabilities, conditions the bus voltage and offers command

to support health monitoring and control[8].

The unit operates in two possible scenarios: the **eclipse phase** and the **sunlight phase**. During the eclipse phase, the spacecraft moves through the shadow of a celestial body (usually a planet) and the solar panels are unable to generate power. In this condition, the battery is the sole power source, discharging to supply the loads while PCDU ensures stable power delivery across the powerlines. During the sunlight phase, the solar panels provide the power. If the generated power exceeds the demand of the active systems, the surplus is used to recharge the battery in preparation of the next eclipse; otherwise, all the generated power is directed to the loads.

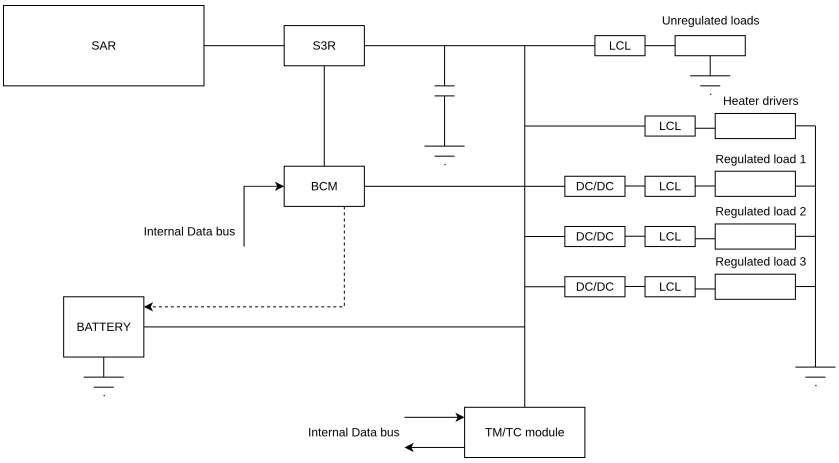


Figure 4.1: PCDU schema

The structural view of this hardware is presented in Figure 4.1 above, defining the following elements:

- **Serial Shunt Switching Regulators (S3R)**.
- **Unregulated and Regulated Bus**.
- **Latching Current Limiter (LCL)**.
- **Heaters**.
- **TM/TC module (Data bus handling)**.
- **Battery Charging Module (BCM)**.

The following subsections present the main electrical components of the PCDU hardware, without going into excessive detail.

4.1.1 Serial Shunt Switching Regulators

As previously established, the **PCDU** is situated between the Battery and the Solar Array, managing the power interface. The Solar Array subsystem inherently generates power independently of the spacecraft's variable load demands. This disparity between the instantaneous power generation and the power required by the main bus can introduce undesirable **voltage and current fluctuations**, potentially compromising system health. To mitigate this critical issue, the most widely adopted solution is the **Serial Shunt Switching Regulator (S^3R)**, also commonly referred to as a **Direct Energy Transfer (DET)** device. Shunt Regulators operate by actively regulating the current drawn from the solar array, thereby ensuring the array's output voltage remains consistently clamped to the required main bus voltage level.

4.1.2 Regulated and Unregulated bus

Loads are connect through power buses, which can be either **regulated** or **unregulated**. The difference between those two implementations lies in the use of controllers to maintain a stable voltage. In an unregulated bus, the voltage depends on the battery characteristic, leading to significant variations between fully charged and fully discharged states. Regulated bus instead provides stable voltage and current making it situable for all those components that are sensible to supply variations. However, the implementation of a regulated bus requires power converters, increasing the complexity and cost of the system.

According to the **European Space Agency (ESA)** [8], the implementation choice depends primarily on the power needed by the loads:

- From about **500W** to **1.5KW**, **28V** is used;
- Up to **8kW** or so, **50V** bus is used;
- For higher power, **100V** bus is used

The decision of the implementation depends also on mission-specific requirements, efficiency constraints and on the sensitivity of onboard subsystems. For example, small satellites often adopt unregulate buses to minimize mass and cost, while large spacecrafts with high-performance electroncics, such as telecommunications satellites, typically employ regulated buses to ensure reliable operations.

In the unregulated bus architecture, loads are connected directly to the battery output.

4.1.3 Battery Charging Module (BCM)

The BCM is linked to both the main bus and to the S3R. It reads the voltage and uses **monitors** to read the charging and discharging current. Based on these measurements, the BCM regulates the S3R to mantain the desired current on the main bus. This module implements control algorithms for both **constant current (CC)** and **constant voltage (CV)** phases.

Beyond the purely regulated and unregulated topologies, **hybrid architectures also exits**, combining elements of both approaches. In such configurations, part of the power is handed through direct battery connection (unregulated), while all the specific loads requiring steady voltage are employed on specific regulated powerline using DC/DC converters.

4.1.4 Latching Current Limiter

The Power Control & Distribution Unit, as expressed before, allows the current to flow to satellite loads. Since it is not redundant, it represents the single point of failure. To prevent failures from propagating to all systems, measures shall be implemented. A circuit able to do it is the **Latching Current Limiter (LCL)** (**??**), it is placed between the loads and the DC/DC converter in the regulated bus. This circuit monitors the amount of current flowing out of the PCDU and, if the amount of current exceed a define threshold, it disconnect the circuit switching to an open state, preventing further current flow.

4.1.5 Heater Drivers

PCDU has also heaters driver (**??**) to turn ON/OFF heaters in all the spacecraft.

Each electronic device has a range of temperature where it performs optimally; therefore, thermal conditions must be carefully managed. During the sunlight phase of the orbit, the satellite's external panels regulated the incoming radiation by reflecting the sunlights, limiting the heat absorption. In contrast, during the eclipse phase, since the outside temperature is closed to the absolute 0, **heaters** are turned on to maintain the internal temperature in the optimal range.

These heaters are regulated by this unit through telecommand (TC) and are able to send back data to check their correctness.

Since this component needs to interact with the on-board computer, it has an interface module with an own **terminal address** that is able to receive and transmit command following the MILBUS standard, it will be discussed directly in the next pages.

4.2 UML implementation

Following the definition of the **PCDU**'s functional requirements—including critical constraints on minimum bus voltage, maximum battery charging current, and the specified terminal address for internal bus communication—the project entered the design phase, starting with a **UML (Unified Modeling Language)** approach using MagicDraw.

Inside the **tasi.sve** terminal, the following two commands were executed:

- `bootstrapper -deps tasi.sve -name tasi.sve.pcdu -output tasi.sve/Src`
  This command creates the `tasi.sve.pcdu` folder inside `tasi.sve/Src`.
- `make -C target/debug run_magicdraw`
  This command launches MagicDraw, already configured to use the **Src** folder inside `tasi.sve` as the project root.

Once MagicDraw is open, it prompts the user to open a folder, which in this case is `tasi.sve.pcdu`. At this point, all dependencies required by the UML module must be added. There are three main dependencies in this project:

- **ECSS and ISIS libraries**: These libraries support models developed according to the ECSS standard. For this project, version 10.3 of the ECSS library was used because it is stable and well-tested. They provide a set of generic simulation models, such as the **Satellite Electrical Network Simulation Model (SENSE)**, which can be reused across different SMP-based simulators. Each model includes interfaces for communication, failure modeling, events, and parameters. The library file is located in the folder **Simulator10_3** on the desktop.
- **Tasi.sve.common libraries**: Developed by Thales Alenia Space, these libraries contain standard-compliant sample models. They accelerate the development of new simulators by providing reusable components. Although the library is present in the **Src** folder, it is not yet linked to the simulator environment, as it only contains sample models.

Once all dependencies have been added, the model development can begin following standard UML methodology. Key concepts such as **inheritance**, **ownership**, **composition**, and **multiplicity** (often represented as numbers on arrows) are applied throughout the design.

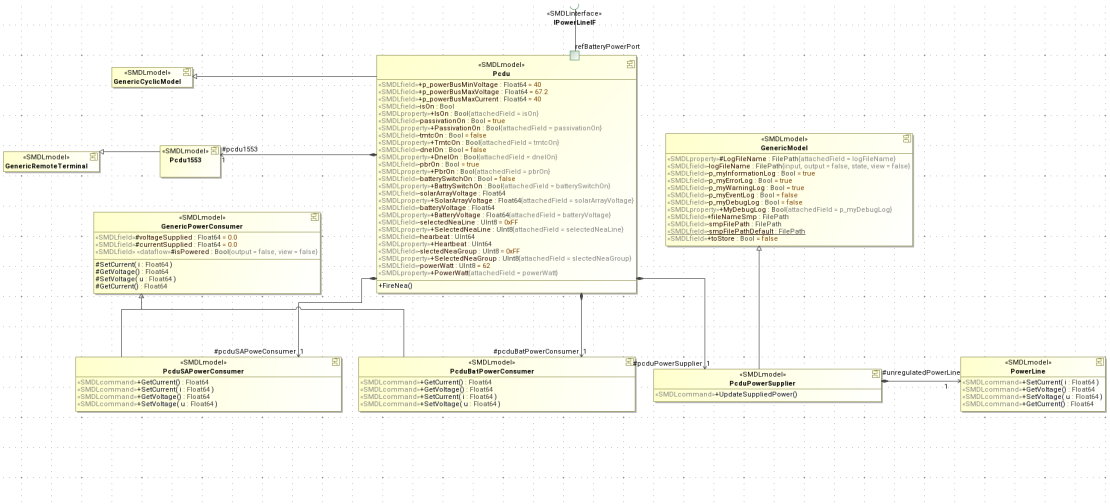


Figure 4.2: PCDU UML

Figure 4.2 represents the UML design of the PCDU. To fulfill the previous explained functionality, this model is composed of several specialized OOP components, each dedicated to a specific task, as listed below:

- `Pcdu`:It represents the main object in the UML model and contains all the data required to configure the key parameters, such as the power needed by the component to operate, the bus voltage range, the battery End-of-Charge Voltage (EOCV), and flags for specific operations (e.g., the firing of the pyro). This component inherits from `GenericCyclicModel`, as it must update its state cyclically every few milliseconds in order to comply with the behavior of the real hardware component.

- `PcduBatPowerConsumer`: It represents the PCDU module that interacts with the battery interfaces. It receives the power generated by the battery and inherits from `GenericPowerConsumer`, as it is modeled as an electrical power consumer. The module implements the methods `GetCurrent`, `SetCurrent`, `GetVoltage`, and `SetVoltage`, which define its interaction with the battery.

- `PcduSAPowerConsumer`: It represents the PCDU module that interacts with the solar array interfaces. It receives the power generated by the solar array and inherits from `GenericPowerConsumer`, as it is modeled as an electrical power consumer. The module implements the methods `GetCurrent`, `SetCurrent`, `GetVoltage`, and `SetVoltage`, which define its interaction with the solar array.

- `PcduPowerSupplier`: It represents the PCDU module that interacts with the output interfaces. It sends to each on loads, the current amount of voltage and

current they need. Since this component provides just few features, it inherits from `GenericModel`. The important functions are `UpdateSuppliedPower`, `SetPortImplementation`, `SetStatusLoad`.

- `Pcdu1553`: It represents the PCDU module responsible for communicating with the onboard computer. It is able to handle the MILBUS messages and responding appropriately to the commands received. This module simulates the behavio of the remote terminal within the PCDU. It inherits from `GenericRemoteTerminal` and the main functions are `BCtoRT`, `caseCmdBcRTRead`, `enableSubAddress`.

In the figure, it can be seen that the components inherit only from the `Generic` models, while all submodels are instantiated inside the `Pcdu` object. This design does not prevent exchanging values between submodels, because the ECSS standard provides each element with common attributes, such as a **pointer** to the element that instantiated it, allowing values to be shared across all simulated models. The UML diagram also defines three power lines, which are not visible in the figure: one connected to the battery, one to the solar array, and one to the generic loads.

In the diagram, it can be seen that all variables are defined as `SMDLfield`, while all methods are defined as `SMDLoperation`. When these definitions are made public and tagged with SMDL, the variables and methods become directly visible in **SIMSAT**. This allows the user to monitor variables in real time or even execute methods while the simulation is running.
After the UML has been drawn, **MagicDraw** allows the user to export it directly in a C++ format compliant with the ECSS standards, clicking on the button on the navbar, having a hierarchical tree as shown in figure below.

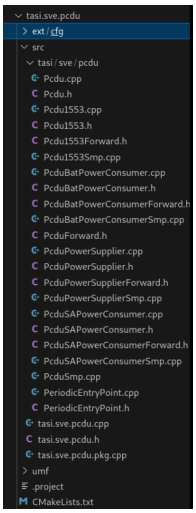


Figure 4.3: PCDU tree hierarchy

It can be observed that for each component, different files are generated, such as `Smp` and `Forward`. The `Smp` file contains the class constructor, while the `Forward` file defines the namespace where the component or its subcomponents are declared. The creation of these files depends entirely on the ECSS-SMP compliant architecture.

At this point of the process, the model is not still linked on the simulator, its just a stand alone project. In order to link models between them, SIMULUS provides a configuration `XML` files where the hierarchy of the entire subsystem with their component is defined. Exists a general XML files where all the XML files of each model is connected to.

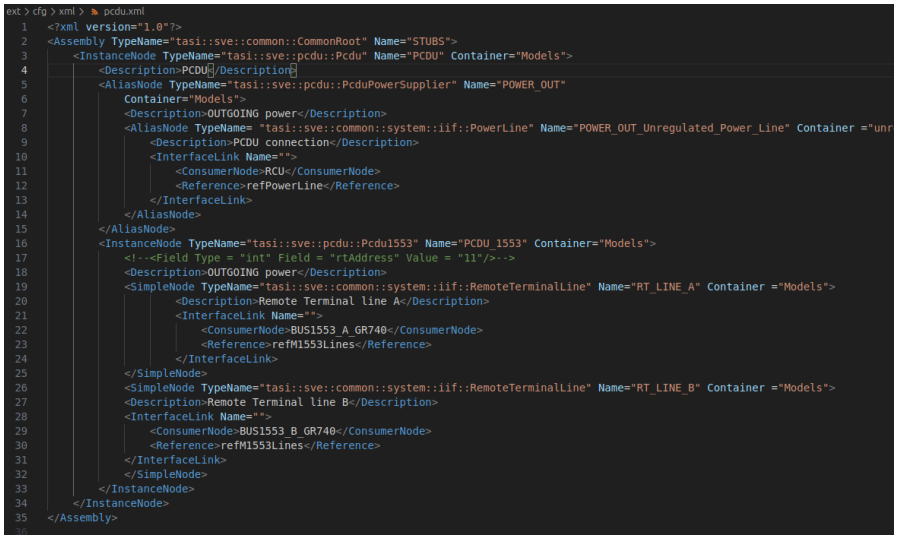


Figure 4.4: PCDU XML

The figure above shows the XML configuration related to the PCDU. The `Assembly` tag specifies the node of the spacecraft hierarchy where the module must be added, in this case `STUBS`. The `InstanceNode` tag defines a component that will be instantiated dring the simulator boot phase. Here, the PCDU module `tasi::sve::pcdu::Pcdu` is instantiated, and its name will be visible in SIMSAT, provided that the path is correct.

The `AliasNode` tag is used to create references to elements that already exist once the simulation is running. For this reason, the `Name` attribute of the `AliasNode` must match the name of the corresponding element instantiated in the `Configure` function. In this case, a reference is created between a `PowerLine` instantiated in in the configuration and the RCU module. The other end of this power line is linked to the `refPowerLine` variable defined in the RCU.

Finally, another `InstanceNode` is used to instantiate the `Pcdu1553` module itself, which groups the remote terminal lines and enables communication between the PCDU and the on-board computer.

The `AliasNode` tag is used to instantiate smaller components within a larger module. In this case, two `SimpleNodes` define the remote temrinal lines (`RT_LINE_A` and `RT_LINE_B`) inside the `Pcdu1553` module. These lines are linked to the on-board computer via the MILBUS, with references to `BUS1553_A_GR740` and `BUS1553_B_GR740`, since the bus is redundant.

In the main `XML` file, the `Pcdu` component is declared directly under `STUBS`. This means that when SIMSAT is running, the PCDU variables can be accessed under the `STUBS` node in the tree displayed on the right side of the GUI.

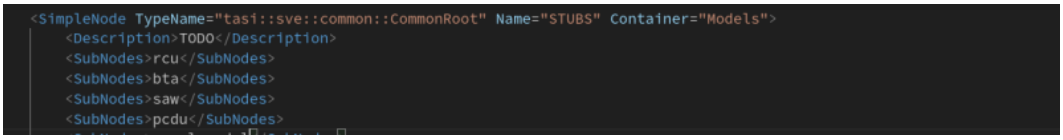


Figure 4.5: Main XML file

In this context, the `Subnodes` tag indicates that the element is not a node itself, but rather a port of the node. The corresponding subnode data are defined within the `InstanceNode` element, whose `Name` attribute matches the value specified in the `Subnodes` tag — in this case, `Pcdu`.

4.3 ECSS-SMP compliancy

The ECSS compliancy is directly visible inside the defined objects since all of them share the same characteristics listed below:

- `Universally Unique Identifier (UUID)`: is a 128 bit long unique identifier, which guarantees unambiguous referencing across simulations and projects. The use of UUIDs avoids naming collisions, simplifies traceability, and ensures that models can be reliably reused and exchanged.
- `Configure method`: initializes the model's parameters, defines operational ranges, and binds external interfaces. Configuration allows the same model template to be adapted for different mission scenarios simply by altering its parameter set. This flexibility is essential to maintain modularity and reduce development cost.
- `Connect method`: establishes links between internal model ports and external data flows, replicating telemetry, telecommand, or sensor pathways present in real spacecraft. The explicit connection mechanism guarantees consistency in data exchange and enforces compliance with interface definitions.
- `Publish method`: allows a variable or method to be visible and run on SIMSAT while the simulator is running.

- `Types`: in the ECSS does not exists the standard variables but each of them is a type of `Smp`, for example `Uint8` is defined as `Smp::Uint8`, giving to it all the properties that it needs to be compliant with these standard.

During the boot phase of each components these methods are called in the following order: `Smp constructor`, `Publish`, `Configure` and `Connect`.

4.4 Pcdu Main Object Implementation

The constructor of the PCDU is defined but not implemented since for debugging purposes has been usefull defined the viariables and their value in a configuration. Its pseudocode is shown below:

```
1: procedure InitializeConstants                          ▷ — Constant Definitions —
2:     const UInt8 rt ← 11                                ▷ for testing purposes
3:     const UInt8 phyAddr ← 18                           ▷ for testing purposes
4:     const Float64 regulatedVoltage ← 28
5:     const Float64 vMean ← 55.0
6:     const Float64 nBattDis ← 0.98
7:     const Float64 nBattCha ← 0.86
8:     const Float64 eocv ← 61.5
9:     const Float64 battResistence ← 25e-3
10:    const Float64 equivalentResistence ← 15.12e-3 + 25e-3
11: end procedure
```

After the call to the constructor, the `Publish` method is invoked to publish all the relevant variables and method in the **SIMSAT** environemt, following the ECSS-SMP standard. Below is shown the pseudocode summarizing the main operations:

Algorithm 1 Publishing Fields

```
1: procedure PublishFields(receiver)
2:     Call base class Publish(receiver)
3:     if requestHandlers is empty then
4:         PopulateRequestHandlers(this, requestHandlers)
5:     end if
                                                          ▷ — Publish Fields —
6:     PublishField("p_powerBusMinVoltage", &p_powerBusMinVoltage)
7:     PublishField("p_powerBusMaxVoltage", &p_powerBusMaxVoltage)
8:     PublishField("p_powerBusMaxCurrent", &p_powerBusMaxCurrent)
9:     PublishField("isOn", &isOn)
10:    PublishField("passivationOn", &passivationOn)
11:    PublishField("tmtcOn", &tmtcOn)
12:    PublishField("dnelOn", &dnelOn)
13:    PublishField("pbrOn", &pbrOn)
14:    PublishField("batterySwitchOn", &batterySwitchOn)
15:    PublishField("solarArrayVoltage", &solarArrayVoltage)
16:    PublishField("batteryVoltage", &batteryVoltage)
17:    PublishField("powerWatt", &powerWatt)
18: end procedure
```

Algorithm 2 Publishing Properties

```
1: procedure PublishProperties                          ▷ — Publish Properties —
2:     PublishProperty("IsOn", Bool)
3:     PublishProperty("BatterySwitchOn", Bool)
4:     PublishProperty("SolarArrayVoltage", Float64)
5:     PublishProperty("BatteryVoltage", Float64)
6:     PublishProperty("PowerWatt", Int32)
7: end procedure
```

The `Configure` method is executed. For each subcomponent, the internal function `common::CommonRoot::CreateContainedModel` is called with the corresponding UUID passed as an argument. The result of this operation is a container, pointed by a variable, whose first element (index 0) holds the instantiated subcomponent. This method instantiates all the previously described subcomponents, and the boot phase for each of them starts again.

In addition, the the `Configure` method defines the `EntryPoints`, which are events triggered automatically, without human intervention. In this case, the entry points are used to increase or decrease the amount of power requested by the PCDU from the solar arrays and the battery. Once defined, the entry points are registered with the scheduler, which integrates them into the simulation environment through the `AddSimulationTimeEvent` function. The following pseudocode summarizes the main operations performed by the `Configure` methond in Pcdu:

```
1: procedure Configure - Pcdu                           ▷ Instantiate all subcomponents
2:     for each subcomponent do
3:         container ← common::CommonRoot::CreateContainedModel(UUID)
4:         subcomponent ← container[0]
5:     end for                                          ▷ Define automatic EntryPoints
6:     for each EntryPoint do
7:         scheduler.AddSimulationTimeEvent(EntryPoint)
8:     end for
9: end procedure
```

Immediately after, the simulator calls the `Connect` method. In this phase, the PCDU is linked to the Solar Arrays and to the Battery according to the XML schema defined for the other components. Once the connections are established, the models can communicate with each other. At this point, the `vBus` variable and the solar array module are initialized to the battery voltage through the `GetVoltage` and `SetVoltage` functions. This initialization ensures that, at simulation time zero, both the bus voltage and the solar array voltage match the battery voltage. The following pseudocode summarizes the main operations performed by the `Connect` methond in Pcdu:

```
1: procedure Connect - Pcdu
2:     solarArray1 ← refSAPort_1
3:     solarArray2 ← refSAPort_2
4:     bta ← refBtaPort
5:     solarArray1.SetVoltage(vBus)
6:     solarArray2.SetVoltage(vBus)                    ▷ Set values at instant 0
7: end procedure
```

4.4.1 PulseRaisingPort

Each `GenericCyclicModel` inherits a function named `PulseRaisingPort` which is a function called multiple times per second, giving to the model the property to change its state in run time having a real time behavior. In this function the PCDU update its state, calling the `Update` function and share with the solar arrays and the loads the new voltage and current values. The following pseudocode summarizes the main operations performed by the `PulseRaisingPort` method:

```
1: procedure PulseRaisingPort - Pcdu
2:     Update()
3:     solarArray1.SetVoltage(vBus)
4:     solarArray2.SetVoltage(vBus)
5:     supplier.UpdateSuppliedPower()                  ▷ Set values at instant 0
6: end procedure
```

In order for the function to be called by the simulator, it is necessary to link it to a clock frequency reference within the simulator by adding the following line to the main `XML` file:

Algorithm 3 Interface Link Definition

```
procedure CreateInterfaceLink
    Set Name ← ""
    Set ProviderNode ← "PCDU"
    Set Reference ← "refClockFrequency_8"
    Set ConsumerNode ← "PULSE_GENERATOR"
end procedure
```

In this case the `Pcdu` model is linked to a `PULSE_GENERATOR` that will call `PulseRaisingPort` every **125 ms** since `refClockFrequency` is equal to 1 second.

The main core is the `Update` function. It calculates the power obtained by the solar array and if it is greater than the load demand, , the excess is sent to the battery for charging during the sun phase, using the CC/CV algorithm and complying with all satellite requirements. During the eclipse phase, the loads rely entirely on the battery. In cases where the solar arrays do not provide enough power for the load demand, the battery supplies the remaining required power.

4.4.2 Battery Charge Module Implementation

This component within the PCDU is responsible for recharging the battery using a Constant Current / Constant Voltage (CC/CV) algorithm, illustrated in Figure 4.6.

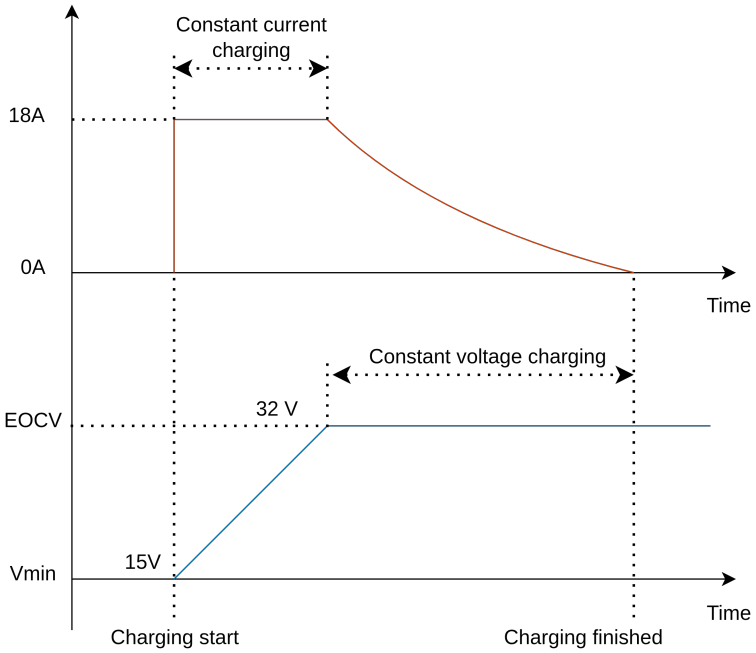


Figure 4.6: Battery charging profile

For this charging scenario to occur, the battery must not be fully charged, and the satellite must be illuminated by the Sun so that the solar array can generate power. In addition, the generated power must exceed the power demanded by the loads, allowing the surplus energy to be used for charging.

During charging, it is essential to ensure that the current flowing into the battery does not exceed its maximum allowable value, as this could cause damage. In this thesis, the CC/CV algorithm is implemented inside the `Update` function of the `Pcdu` component using the following variables:

$$I_{residual} = I_{solarArray} - I_{required} \tag{4.1}$$

$$I_{maximum} = 42.0\ A \tag{4.2}$$

$$I_{maximumCurrentAtVoltage} = \frac{EOCV - OCV}{R_{equivalent}} \tag{4.3}$$

Here, $I_{maximumCurrentAtVoltage}$ represents the maximum current that the battery can safely receive at a given voltage. When the battery is deeply discharged, this value equals the maximum current it can accept. As the battery approaches full charge, the allowable current decreases significantly. If the residual current exceeds 42.0 A but the battery is nearly charged, the charging current is limited to $I_{maximumCurrentAtVoltage}$ rather than the fixed maximum. Conversely, if the residual current is below this value, only that available amount is used for charging. At each iteration, the algorithm selects the minimum between these two values to determine the actual charging current.

In this context:

- $EOCV$ is the end-of-charge voltage, i.e., the maximum voltage the battery can reach;
- OCV is the open-circuit (current) voltage of the battery;
- $R_{equivalent}$ represents the equivalent resistance between the battery and the PCDU.

4.4.3 Unregulated Bus Implementation

The unregulated bus implementation is based on a function that depends on two main factors: the open-circuit voltage (OCV) and the current flowing into or out of the battery. The relationship is expressed by the following equation:

$$V_{Bus} = V_{OCV} + I_{toBattery} \times R_{equivalent} \tag{4.4}$$

It should be noted that $I_{toBattery}$ varies at each simulation step. As the battery discharges, the current flowing out of it increases, while during charging, the incoming current increases instead. Consequently, the bus voltage (V_{Bus}) continuously changes throughout the simulation.

During the eclipse phase, $I_{toBattery}$ is negative because current flows out of the battery, resulting in a bus voltage lower than the battery voltage at that moment. Conversely, during the sunlight phase, the current is positive, making the bus voltage higher than the battery voltage. When the battery is nearly discharged, the voltage difference between the battery and the bus reaches its maximum, as the discharge current is limited (clamped) to 42.0 A.

The following pseudocode represented how the previous concepts has been implemented.

```
1: procedure UPDATE - PCDU                                          ▷ Update the internal Pcdu State
2:     btaVoltage ← bta.GetVoltage()
3:     Psa ← solarArray.GetVoltage() * solarArray.GetCurrent()
4:     Psc ← powerWattPcdu + totalPowerAmount
5:     currentToSendToBattery = 0          ▷ Power generated by Solar Arrays and power needed by the loads
6:     if Psa ≥ Psc then
7:         mode ← SolarOnly
8:     else if Psa ≠ 0.0 then
9:         mode ← BatteryOnly
10:    else
11:        mode ← SolarPlusBattery
12:    end if
13:    switch mode
14:        case SolarOnly
15:            requiredCurrent ← Psc / vBus
16:            remainingCurrent ← solarArray.GetCurrent - requiredCurrent
17:            maximumCurrentAtVoltage ← (EOCV - btaVoltage) / equivalentResistance     ▷ If there is more
   current than required, send the current to the battery
18:            if remainingCurrent ≥ 0 and btaVoltage ≤ EOCV  then
19:                currentToSendToBattery ← Min(maximumCurrentAtVoltage,remainingCurrent,42.0) ▷ Current
   towards the battery shall follow the CC/CV algorithm and shall be less than 42.0 A
20:                vBus ← btaVoltage + equivalentResistance * currentToSendToBattery
21:                bta.SetCurrent(currentToSendToBattery)
22:            else
23:                vBus ← btaVoltage
24:                bta.SetVoltage(0.0) ▷ In case there is no enough current to send to the battery, it is equal to 0
25:            end if
26:        end case
27:        case BatteryOnly
28:            if vBus ≥ minBusVoltage  then
29:                currentToSendToBattery ← - Psc / vBus
30:                if btaVoltage + equivalentResistance * currentToSendToBattery ≥ powerBusMinVoltage  then
31:                    vBus ← btaVoltage + equivalentResistance * currentToSendToBattery
32:                else
33:                    vBus ← minBusVoltage
34:                    currentToSendToBattery ← (vBus - btaVoltage) / equivalentResistance
35:                end if
36:                bta.SetCurrent(currentToSendToBattery)
37:            end if
38:        end case
39:        case SolarPlusBattery
40:            to do
41:        end case
42:    end switch
43: end procedure
```

It is important to note that this function is executed every **125 ms**. At each iteration, it calculates both the power generated by the solar array and the power demanded by the loads. When the solar array produces power (i.e., the satellite is in the sunlight phase), the system prioritizes using this power over drawing energy from the battery. This behavior is managed through the conditional statements within the function.

The implementation also includes checks for special conditions, such as when the battery is fully charged or discharged. As shown in the pseudocode, the bus voltage (`vBus`) is updated in all operating modes. Additionally, a case is defined for situations where the solar array alone cannot meet the load demand—in such

scenarios, the battery supports the power supply. This last mode represents an edge condition not fully explored in this thesis but included for potential future development.

4.5 Milbus Development

The file named `Pcdu1553` implements the **MIL-STD-1553 bus** interface and handles the received messages. Its boot phase begins when it is instantiated in the configuration phase by the main component, as previously described. Since this element is relative simple, the `Configure` method call just a function named `enableSubAddress` which turn on all the subaddresses of this model and the pseudocode is shown below.

Algorithm 4 Enabling All Subaddresses in Pcdu1553

```
1: procedure EnableSubAddress - Pcdu1553        ▷ Enable all subaddresses from 1 to 30 and set directions
2:     for subAddress ← 1 to 30 do
3:         tmpSubAddrConf ← subAddressMap[subAddress]
4:         tmpSubAddrConf.isManaged ← true
5:         tmpSubAddrConf.en.directionSaToBc ← true
6:         tmpSubAddrConf.en.directionBcToSa ← true
7:         tmpSubAddrConf.en.enableSubAddress ← true
8:         tmpSubAddrConf.enDefault.directionSaToBc ← true
9:         tmpSubAddrConf.enDefault.directionBcToSa ← true
10:        tmpSubAddrConf.enDefault.enableSubAddress ← true
11:        subAddressMap[subAddress] ← tmpSubAddrConf
12:    end for
13: end procedure
```

The `Connect` methods is left empty since the connection to the Milbus physical bus is established directly from the `XML` file of the Pcdu.

Before starting with the explaination of the implementation is relevent introduce the Milbus and its properties.

4.5.1 Milbus Overview

Figure 4.7 shown the architecture of the Milbus protocol and defines only three type of terminals:

- **Bus controller (BC)**: it's the **master** in the protocol and is main function is to control **all** the communications on the bus. It initiates the data transfers following a command/response schema - the BC sends a commant to the RTs, which reply with a response. Are allowed more than one BC on the same architecture but only one can be active at a time, others can be implemented for redundant purposes. In this thesis context, the Bus Controlled is the On-board Computer, emulated using TEMU.

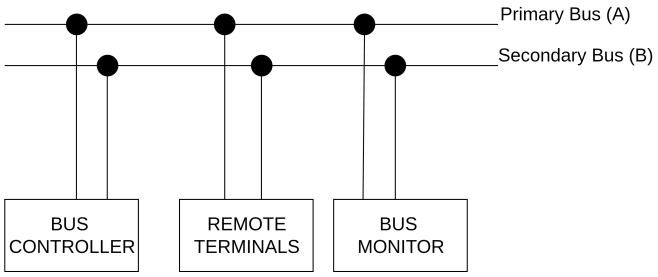


Figure 4.7: milbus architecture

- **Remote Terminal (RT)**: it's the **slave** in the protocol and **always** answer to a message from the BC. An RT represents any subsystems that needs to exchange data using the 1553 data bus. This protocol is plug and play, since if the device does not natively support it, an interface unit can be used to translate messages and connect the device to the BC. In this thesis context, the Remote Terminal is the PCDU since it needs to communicate with the On-board Computer

Messages are made by **words**, which are the smallest unit that can be transmitted over the bus. The protocol defines three type: **command**, **data** and **status**.

Command Word

Remote terminal address (0-31)					Receive or Transmit	Location (sub-address) of data (1-30)					Number of words to expect (1-32)				
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 4.8: Command word bit usage

The Command Word (Figure 4.8) consists of the following fields: first 5 bits identify the remote terminal receiving. The sixth bit indicates the direction of the transfer, 0 for receiving and 1 for transmitting. Bits from seventh to eleventh specify the sub-address that indicate where store or get data on the terminal. Last 5 bits indicate how many words to expect after the command word, the limit depends on how many values you represent with 5 bits, which is from 0 to 31.

4.5.2 Milbus Implementation

The class inherits from `GenericRemoteTerminal`, which provides the `BCtoRT` function. The `BCtoRT` function is invoked directly from the simulator environment whenever the on-board computer sends a MIL-STD-1553 message to the remote terminal, which is divided in *data* (a sequence of byte out of the protocols architecture) and *command* (having the same structure of the status word explained

before). The following pseudocode summarizes the main operations performed by the `BCtoRT` method:

```
1: procedure BCtoRT - Pcdu1553(commandW, data, exchangeStatus,exchangeEpochTime,exchangeSimTime)
2:     switch mode
3:         case CMD_BROADCAST_READ
4:             to do
5:         end case
6:         case CMD_READ caseCmdBcRtRead(data, commandW)
7:         end case
8:         case MODE_BROADCAST_READ
9:             to do
10:        end case
11:    end switch
12: end procedure
```

This switch defines the logic required to handle the different types of messages supported by the MIL-1553 protocol. Within the scope of this thesis, the main objective is to enable the reception of a MIL-1553 command to turn a generic load on or off. For this reason, only the `CMD_READ` case has been implemented. It is important to highlight that MIL-1553 communication calls are managed directly by the emulator, and that the variable names and command structures are defined in compliance with the ECSS standard.

This implementation serves as a foundation for future extensions, where additional message types (e.g., broadcast commands or mode-specific operations) could be supported to simulate a more complete MIL-1553 communication interface.

When a read message is received, the `caseCmdBcRtRead` function is triggered. This function analyzes the data field, extracting the load identifier and its corresponding status. The function invokes the `SetStatusLoad` method in `PcduPowerSupplier` class. The following pseudocode summarizes the main operations performed by the `caseCmdBcRtRead` method:

```
1: procedure caseCmdBcRtRead - Pcdu1553(data, commandW)
2:     cmd ← data & 0x7F
3:     turnOnOrOff ← (data ≫ 7) & 0x01
4:     load ← (data ≫ 8) & 0xFF                    ▷ Retrieve information from the data field
5:     Print("Subaddress:", commandW.subAddress)
6:     Print("turnOnOrOff:", turnOnOrOff)
7:     Print("Load:", load)
8:     switch commandW.subAddress
9:         case SA_2                    ▷ The Switch act as a filter based on the command subaddress
10:            switch load
11:                case 0
12:                    supplier.SetStatusLoad(Loads(load), Status(turnOnOrOff))
13:                end case
14:            end switch
15:        end case
16:    end switch
17: end procedure
```

This function performs the direct interpretation of the MIL-1553 message contained in `data` using the associated `commandW`. Within the scope of this thesis, it is sufficient to handle a single dummy message. For this purpose, only messages with a subaddress equal to 2 are processed, and the data field is restricted to a single byte.

In this byte, the most significant bit indicates the command type: if it is set to 1, the corresponding load is turned on; if it is 0, the load is turned off. The remaining bits are used to identify which load in memory should be affected. This implementation demonstrates the basic mechanism for decoding and acting on MIL-1553 messages, while serving as a foundation for handling more complex or multiple-message scenarios in future work.

4.6 Pcdu Outgoing Interface Implementation

The class named `PcduPowerSupplier` implements the Pcdu supply interfaces. Loads are identified by an enumeration, and each load is associated with a status. The `SetStatusLoad` method updates the status of a specified load, provided it exists in the internal collection. The status is expressed using the dedicated `Status` boolean value. The following pseudocode summarizes the main operations performed by the `SetStatusLoad` method:

```
1: procedure SetStatusLoad(load, status)
2:     loadPowerLine ← loads.find(load)
3:     if loadPowerLine = loads.end() then
4:         return                                          ▷ Load not found
5:     end if
6:     loadPowerLine.status ← status                       ▷ Set ON or OFF
7: end procedure
```

In this function, the object maintains a static collection `loads`, where each load is identified by an enum and its corresponding status is stored as a separate value. If the specified load is found in the collection, its status is updated to the new value, effectively turning the load ON or OFF. This mechanism ensures that load states are managed consistently and can be accessed or modified throughout the simulation.

This class also provides the `UpdateSuppliedPower` method, which is invoked by `Pcdu` to update the internal states of the active loads. The PCDU keeps track of which loads are switched on. If a load is connected to a regulated bus, it receives a fixed regulated voltage; otherwise, it receives the instantaneous Vbus voltage. This mechanism ensures thatthe satellite simulator behaves as a real-time system, where all internal states are continuosly updated at each simulation step. The following pseudocode summarizes the main operations performed by the `UpdateSuppliedPower` method:

```
1: for each load in loads do
2:     switch load.status
3:         case ON
4:             if load.voltageType = REGULATED then
5:                 voltageToSupply ← regulatedVoltage
6:             else
7:                 voltageToSupply ← batteryVoltage
8:             end if
9:             current ← load.powerNeeded / voltageToSupply
10:            update load with current and voltage
11:        end case
12:        case OFF
13:            update load with 0 voltage and 0 current
14:        end case
15:    end switch
16: end for
```

Each load has an associated enum specifying its voltage type, as defined in the configuration file described earlier. If a load is ON and connected to a regulated bus, it receives a fixed voltage equal to the regulated voltage; otherwise, it receives the current bus voltage. The current supplied to the load is calculated as the required power divided by the voltage it receives. If a load is OFF, both the supplied voltage and current are set to 0. This structure ensures that each load receives the correct power according to its state and voltage type throughout the simulation.

4.7 Powerline Implementation

All the module developed on simsat are independent module and in order to communicate they could implement several kind of solutions. SMP supports three different methods for components communication: direct interface, data flow and event based.

In the first one, a model has several interfaces that establish an agreement between the others. Every model implementing the interface has all the functionalities it provides, defining a **Consumer** and a **Provider**. In this thesis context this approach has been used.

The figure above illustrates a unidirectional powerline architecture. In this setup, communication occurs in only one direction: Model B can invoke methods defined in Model A to retrieve data or set new values. This functionality is enabled by the powerline interface defined in the ECSS standard. Model A cannot directly communicate with Model B; a separate powerline in the opposite direction would

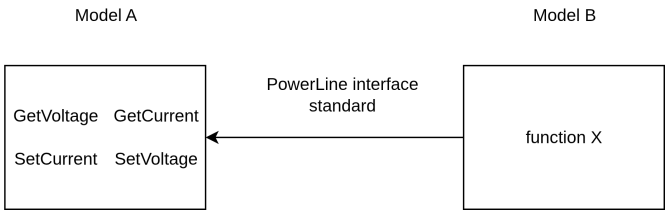


Figure 4.9: powerline architecture

be required for bidirectional interaction.

In the context of this thesis, Model A corresponds to the battery, since the PCDU sets its current and uses a `Get` method to read the voltage. Similarly, the `PcduPowerSupplier` also functions as Model A for all satellite loads, as they set the current and retrieve the voltage from it.

In order to implement this architecture, the model A shall define a method name `SetPortImplementation` having the following pseudocode:

```
1: procedure SetPortImplementation - PcduPowerSupplier
2:     powerPortImplementation.getCurrentPort ← Bind(GetCurrent, this)
3:     powerPortImplementation.setCurrentPort ← Bind(SetCurrent, this, placeholder)
4:     powerPortImplementation.getVoltagePort ← Bind(GetVoltage, this)
5:     powerPortImplementation.setVoltagePort ← Bind(SetVoltage, this, placeholder)
6:     ConnectPort(PowerLine, PortImplementation, Unregulated_Power_Line, powerPortImplementation, true)
7: end procedure
```

In this function, the `powerPortImplementation` interface maps its methods using the C++ `Bind` function to the corresponding getter and setter methods of the object. Once mapped, the port is connected to one end of an unregulated power line defined in the ISIS model. For the powerline to operate correctly, a corresponding module must exist at the other end.

The counterpart is defined in the XML file of Model A using an alias node:

```
1: procedure CreateAliasNode - POWER_OUT_Unregulated_Power_Line
2:     TypeName ← "tasi::sve::common::system::iif::PowerLine"
3:     Name ← "POWER_OUT_Unregulated_Power_Line"
4:     Container ← "unregulatedPowerLine"
5:     Description ← "PCDU connection"
6:     ConsumerNode ← "RCU"
7:     Reference ← "refPowerLine"
8: end procedure
```

This alias node serves as the endpoint for the powerline, ensuring that data flows correctly between the PCDU and the connected loads or modules (in the examples is named as `RCU`). It defines the consumer node and reference, allowing the PCDU to provide current and voltage to the connected elements through the unregulated power line.

The model B in its `Connect` method must connect its part to the cable, as `Pcdu` did for the battery and solar arrays.

It is important to note that the connections between models are established within the `Connect` function. Therefore, all calls to `SetPortImplementation` must be executed beforehand during the `Configure` phase. The following pseudocode illustrates this process in the `PcduPowerSupplier` object.

```
1: procedure Configure - PcduPowerSupplier(logger, linkRegistry)   ▷ Create the unregulated power line model
2:     model ← CreateContainedModel("_Unregulated_Power_Line", this, unregulatedPowerLine, UUID_PowerLine)
3:     assert(model)
                                                                   ▷ Store interfaces for later use
4:     unregPowerLineInterface ← model casted to IPowerLineIF
5:     unregPowerLine ← model casted to PowerLine
                                   ▷ Bind getters and setters to the power port before connecting models
6:     SetPortImplementation()
                                          ▷ Call base class Configure last to perform state transition
7:     GenericModel::Configure(logger, linkRegistry)
8: end procedure
```

4.8 Battery and Solar Array Interface

At the beginning of the thesis, the interfaces for connecting the modules were defined in `Pcdu`. These objects are implemented using a consistent approach: each has `Get` and `Set` methods that internally call the corresponding getters and setters of the `Pcdu` object.

Below are pseudocode snippets of some of these methods, included for reference and potential use in future extensions.

```
1: procedure IsCharging - PcduBatPowerConsumer
2:     return charging          ▷ Return true if the battery is currently charging, false otherwise
3: end procedure
```

```
1: procedure SetCharging - PcduBatPowerConsumer(value)
2:     charging ← value                          ▷ Set the charging state to true or false
3: end procedure
```

```
1: procedure StopCharging - PcduBatPowerConsumer
2:     charging ← false                                  ▷ Stop charging the battery
3:     currentSupplied ← 0                    ▷ Reset the current supplied to zero
4: end procedure
```

Chapter 5

Validation and Verification

The verification and validation (V&V) phase is essential to ensure the reliability of the developed simulator. The verification process addresses the question *"Are we building the system right?"*, by ensuring that the implementation adheres to the defined design specifications and requirements, while the validation process focuses on determining whether the system accurately reproduces the expected real-time behavior. In the context of the PCDU and power system simulator, V&V activities ensure that the implemented algorithms (e.g., power distribution, CC/CV battery charging, regulated/unregulated bus management) are correctly realized in software, that the internal states (voltages, currents, and power flows) evolve consistently with the underlying models, and that the outputs match expected behaviors under nominal conditions, boundary cases, and fault scenarios. Furthermore, the validation has been performed by comparing the simulator's results against the official power budget provided by **TASI** for the IOS mission, ensuring that the developed model aligns with realistic mission-level power profiles. This phase therefore provides confidence that the simulator can be reliably used as a test and analysis tool for satellite subsystems.

In-orbit servicing satellites were conceived to extend the operational life of spacecraft already in orbit. A satellite's lifespan is heavily constrained by its fuel reserves and its inability to receive repairs once deployed. In-orbit servicing addresses these limitations by enabling a dedicated spacecraft to approach an existing satellite, dock with it, and perform maintenance tasks. Depending on the mission objectives, the servicer can refuel the client satellite, correct malfunctions, or restore functionality when performance deviates from expectations.

In the following paragraph, a set of simulation is presented to support the verification and validation activities. The graphs illustrated the behavior under different operating conditions, allowing the comparion with the expected system response. For each mission phases are reported the following data:

- **Power Generated from Solar Arrays**
- **Power loads request**
- **Battery charge/discharge current**
- **Main bus (unregulated +49/+63) voltage**
- **Battery State of Charge (SOC)**

5.1 From Launch to In-Orbit Operations: Mission Phases

In the first experiment are simulated different mission phases:

- **Ground and Launch**: lasts **4760** seconds
- **Launcher Separation to SA deployment**: lasts **3181** seconds
- **SA deployment to target release - starting in sunlight**: lasts **6144** seconds
- **In-orbit phase (S-band Tx only)**:lasts 10 orbital cycles, **5520** seconds
- **In-orbit phase(S-band + X-band Tx)**:lasts 10 orbital cycles, **5520** seconds

As shown in the solar arrays power figure, during ground and launch phase and the launcher separation to SA deployment, the power is supplied only by the on-board battery, reason why the power generated by the solar array is equal to 0. 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 battery during the eclipse phase The red markes in the figure indicate the transition points between one mission phase and the next. The variations in power demand arise from different combinations of loads being activated or switched off across the mission phases. These specific loads are not detailed here for confidentiality reasons. In the plots, the solar incidence angle is 7° during the deployment of the solar arrays, while a value of 30° is used for the in-orbit phases.

Before commenting this graphs, is important describe what is the State of Charge (SOC). It quantifies the remaining capacity available in the battery. It is defined as the ratio of the remaining charge in the battery, divided by the maximum charge that the battery can deliver. The result is expressed as a percentage. Even when the battery is not being used, its SOC decrease over time. It's known as self-discharge In the SOC plot of Figure 5.2, the state of charge decreases as expected during the initial mission phase, since the solar array is not yet generating power. As time

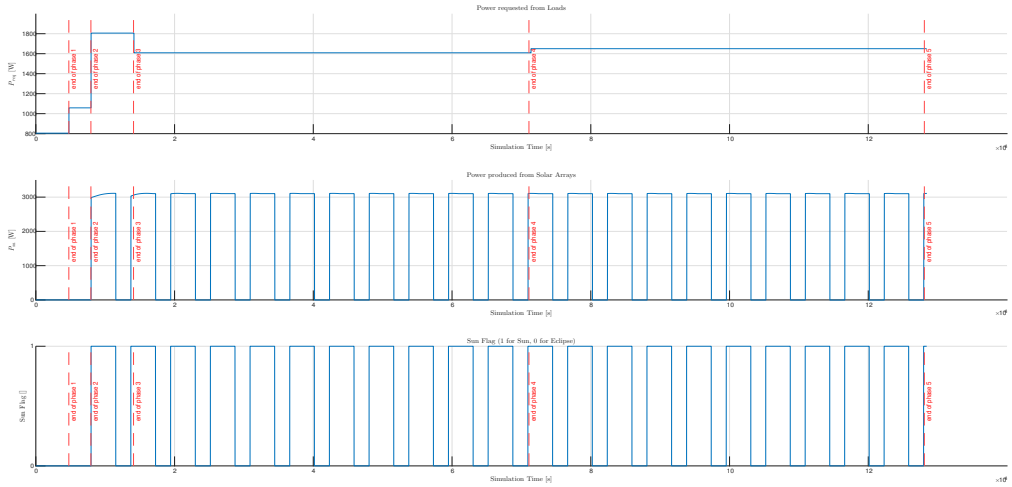


Figure 5.1: Loads power, SA power, sun/eclipse flag

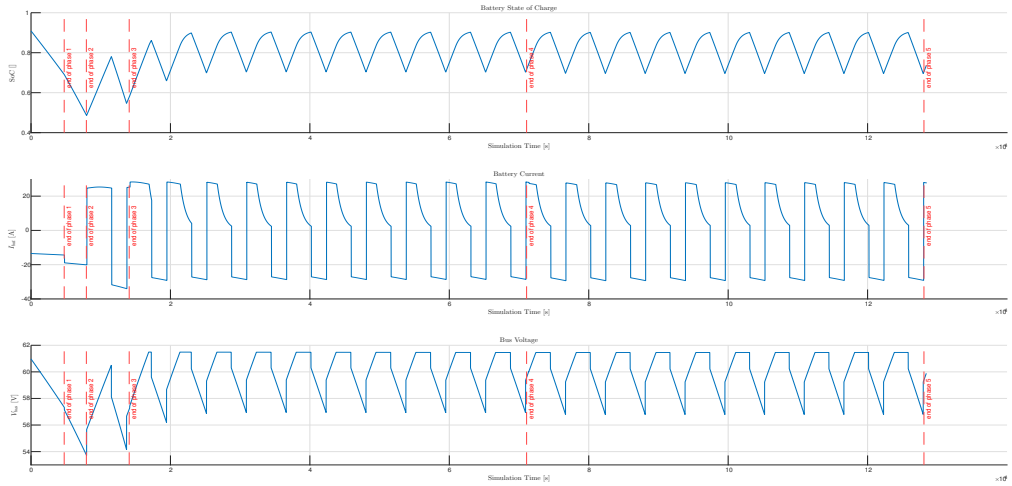


Figure 5.2: SOC, battery current and bus voltage

progresses, the behavior of the CC/CV charging algorithm becomes apparent. At the start of each sunlight period, the charging current to the battery is higher, while towards the end it decreases, according to:

$$I_{battery} = \min\left(\frac{EOCV - OCV}{R_{eq}}, \frac{P_{sa} - P_{sc}}{V_{bus}}\right) \tag{5.1}$$

As expected, the bus voltage V_{bus} does not coincide with the battery voltage. Instead, it follows:

$$V_{bus} = OCV_{Batt} + R_{eq} \cdot I_{battery} \tag{5.2}$$

During sunlight phases, the current flowing into the battery is positive, resulting in a bus voltage higher than the open-circuit voltage. This corresponds to the step-up visible in the plot. When transitioning to eclipse, the current decreases or reverses, causing V_{bus} to step down.

Finally, since the algorithm follows the CC/CV method, once the battery reaches full charge and no current flows into it, V_{bus} settles at the battery end-of-charge voltage.

5.2 Rendezvous Phases

In the second experiment, a rendezvous scenario is simulated, in which the IOS satellite begins its approach to the target during an eclipse period. The sequence is divided into the following phases:

- **Start of target approach**: no fixed duration.
- **Target recognition**: lasts **600** seconds.
- **Docking phase in close range**: lasts **300** seconds.
- **Capturing phase**: lasts **1200** seconds.

The rendezvous takes place entirely during the eclipse period, when no power would be generated by the solar arrays regardless. Since the battery starts fully charged from the previous mission phase, this scenario represents the best possible case.

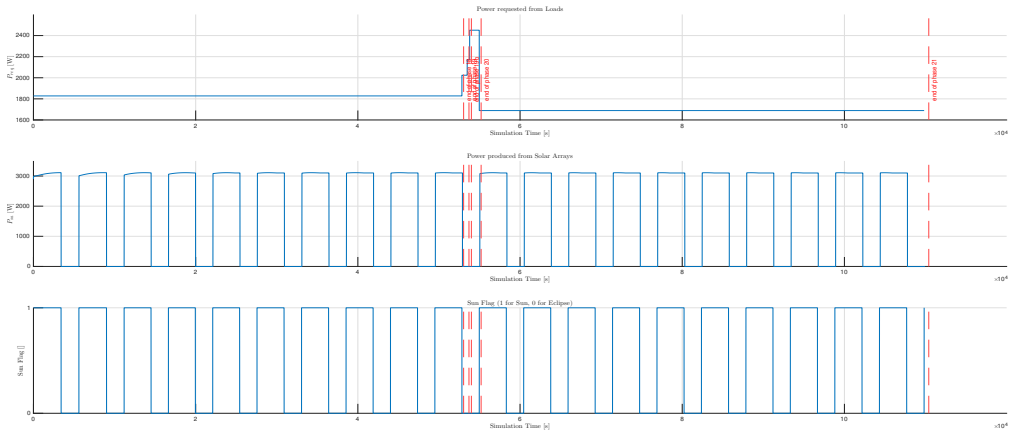


Figure 5.3: Loads power, SA power, sun/eclipse flag

In this case, the entire satellite relies on the battery throughout the operation. Unlike the previous experiment, the power demand is higher, as this scenario represents the mission's core activity and requires more subsystems to be active.

The **target approach** begins while the battery is still almost fully charged, making it the most favorable starting point for an eclipse phase. It is important to note that, during the power demand peak, the bus voltage approaches its minimum operating threshold without crossing it. At the same time, the current drawn from the battery increases as expected.

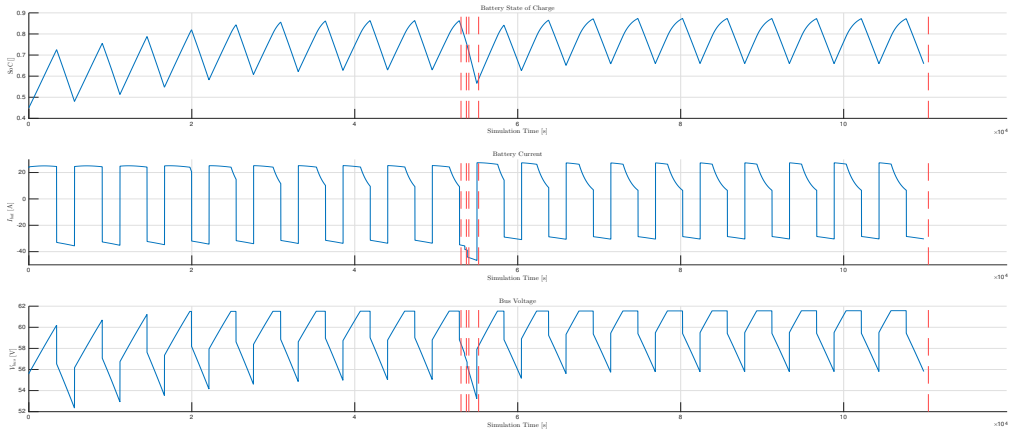


Figure 5.4: SOC, battery current and bus voltage

These results show that the considered mission phases can be sustained under the assumed onboard conditions.

5.3 Ground Segment Testing

The simulation environment consists of two virtual machines connected through the same NAT network: a **SUSE VM**, running the spacecraft simulator (`SIMSAT.sh`), and a **Windows VM**, referred to as ECHO, which serves as the ground station.

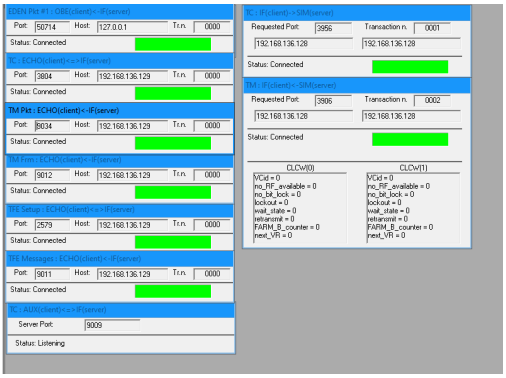
The setup begins by launching the SIMSAT process on the SUSE VM without starting the simulation. This ensures that the simulator is ready to accept connections from the ground station interface. On the ECHO VM, the ground station interface (`iride_if.exe`) is configured with the simulator's IP address, the satellite identifier corresponding to the simulated spacecraft, and the communication ports defined in the SIMSAT configuration. Once configured, the interface establishes TCP/IP connections for both telecommands (TC) and telemetry (TM), and the connection status is confirmed by logs on the simulator.

Following this, the ground station mission operations software is launched, enabling the management of mission setup, telemetry monitoring, and session initialization. Additional monitoring applications capture telemetry data and confirm that the communication links with the simulator are active. This preparation ensures that all subsystems, including the on-board PCDU, can respond to

telecommands as expected.

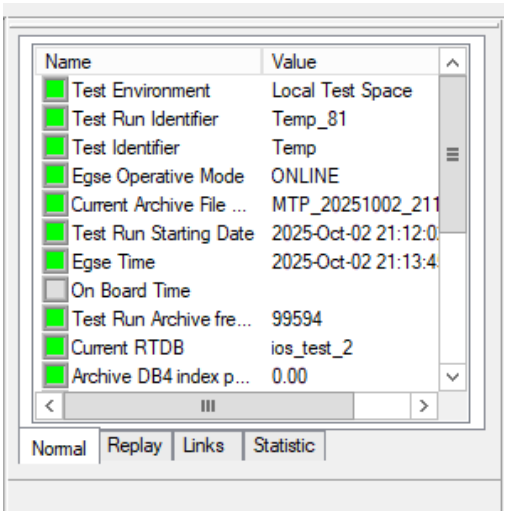
With the connections established, the simulation is started from SIMSAT. Telemetry generated by the on-board software is displayed both in SIMSAT and the ground station consoles, allowing verification of data flow and system behavior. Predefined test telecommands can be issued to validate the communication links and confirm proper reception and execution by the simulated spacecraft.

The **telecommand control (TCC) module** provides mission control capabilities, including sending telecommands, switching session modes, and monitoring execution results. Custom telecommands can be sent through TCC, and their progress is tracked via telemetry packets that indicate reception, validation, execution, and acknowledgment stages. This closed-loop setup ensures that both nominal operations and custom test scenarios can be executed, monitored, and validated in a controlled environment. simulator runs on a SUSE virtual machine, while a second virtual machine, called Echo, is used as the ground station. Both machines are part of the same NAT network and therefore share the same network segment. SIMSAT has a process listening on a specific port, which allows the SUSE machine to accept a TCP/IP connection from Echo. Once the connection is established, Echo can send PUS commands to the SUSE machine. These commands are then forwarded to the on-board software, which interprets them. If the PUS command targets an internal load, a MILBUS message is sent to the corresponding subsystem—in this case, the PCDU.

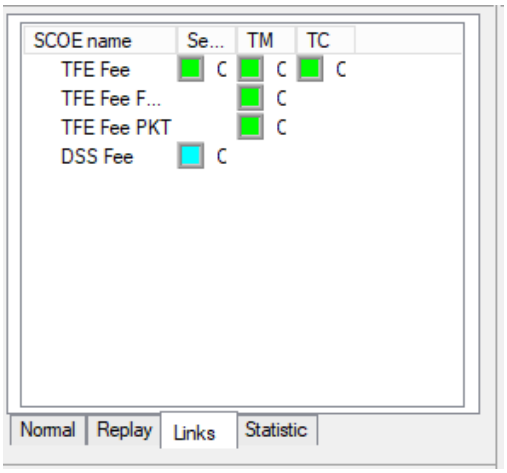


(a) Echo connection view

(b) SIMSAT connection view



(c) Prova 1



(d) Prova 2

Figure 5.5: Overview of simulator and ground station connection views. Top row shows the ground and simulator interface views, bottom row shows internal system displays.

Once the connection is established, ECHO is able to send telecommand to the simulated spacecraft. and from the TCC console can be seen the result as shown in figure below:

In the Figure 5.6, four telemetry packets transmitted from the on-board software to the ground station are shown, namely **TM(1,1)**, **TM(1,2)**, **TM(1,3)**, and **TM(1,7)**. These packets report the status of telecommands sent from the ground station to the on-board computer (OBC):

- **TM(1,1)** indicates that the telecommand has been successfully received by

the OBC and has passed validation.

- **TM(1,2)** indicates that the telecommand was received by the OBC but failed validation. This packet may include additional information explaining the reason for the failure.
- **TM(17,2)** indicates the answer to the specific command sent, containing its response.
- **TM(1,7)** indicates that the execution of the command has been successfully completed.

These telemetry packets provide a clear sequence of the telecommand lifecycle, from reception and validation to execution, allowing the ground station to monitor the status and outcome of each command.

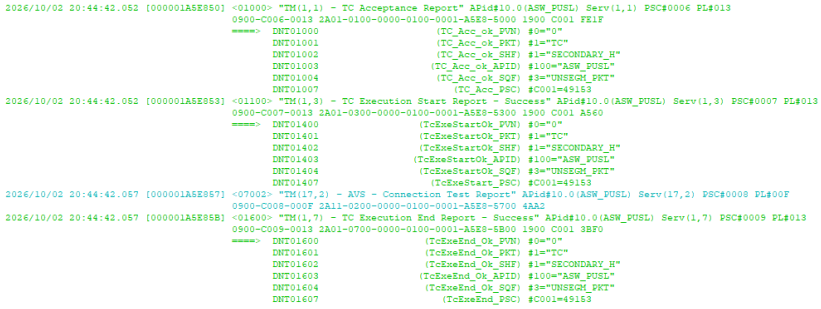


Figure 5.6: TCC after command sent

At this point a raw pus packet can be sent, but before desribing the chosen byes is important give a brief introduction to the protocol.

5.4 PUS overview

All telecommand packets shall conform to the structure shown in Figure 5.7 below.

Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Application Data	Spare	Packet Error Control (see Note 2)
Version Number (=0)	Type (=1)	Data Field Header Flag	Applica-tion Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Figure 5.7: Telecommand structure

It is divided in Packet Header of 48 bits and in Packed data field with variable lenght, it is decided during the mission design.

- **Version Number**: This field is always equal to 0. By changing it, in future more variations on packet usability can be introduced.
- **Type**: This bit distinguishes between telecommand and telemetry packets. Is equal to **1** for telecommand, otherwise **0**.
- **Data Field Header Flag**: This indicates the presence of absence of a data field header. All packets shall have it because it and it shall be set to **1**.
- **Apllication Process ID (APID)**: The APID corresponds to the on-board application process which is the destination for this packet. Each mission has its own list of possible APIDs.

CCSDS Secondary Header Flag	TC Packet PUS Version Number	Ack	Service Type	Service Subtype	Source ID	Spare
Boolean (1 bit)	Enumerated (3 bits)	Enumerated(4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Enumerated (n bits)	Fixed BitString (n bits)

Figure 5.8: Data field header

The data field header contains information about the service and the subservice, allowing the Ground Station to have granular access to the satellites loads.

Giving this information the command sent are shown in the following figures:

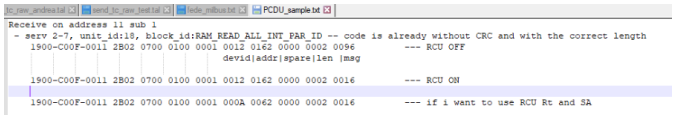


Figure 5.9: PUS string

The last byte of the message, `96` or `16`, is interpreted by the `Pcdu1553` when it receives a Milbus packet. PUS packets do not always carry a Milbus packet directly; instead, the onboard software maintains a list of Milbus packets to transmit over the internal bus when a PUS packet is received. If the PUS packet corresponds to a known Milbus packet, the software sends it over the satellite bus. A specific PUS packet—Service 2 with Subservice 7—allows the PUS packet to carry Milbus data directly. For example, this can be compared to sending data over Ethernet without passing through the other protocol layers.

Chapter 6

Conclusion

The primary objective of this thesis was to design and implement a scalable and reliable simulator for the Electrical Power Subsystem (EPS) of a satellite, integrated into the SIMULUS framework. This objective has been successfully achieved. The simulator reproduces the behavior of the EPS, including the battery, solar arrays, and the Power Control and Distribution Unit (PCDU), as well as the interface with the On-Board Computer (OBC) through the MIL-STD-1553B bus and the Packet Utilization Standard (PUS). By integrating these components, the simulator allows for accurate assessment of satellite power management under different operational scenarios, including nominal, boundary, and fault conditions.

The verification and validation (V&V) activities presented in Chapter 5 confirmed that the simulator operates according to the specified design requirements. The implemented CC/CV charging algorithm correctly regulates the battery current, while the bus voltage behaves consistently with the expected system dynamics. Simulated mission phases demonstrated the capability of the EPS to sustain power demands during both sunlight and eclipse periods, including complex scenarios such as rendezvous operations. These results provide confidence that the simulator can serve as a reliable tool for testing, validation, and optimization of spacecraft power systems before launch.

One of the key contributions of this work is the scalable architecture of the simulator. The modular design allows for future extensions, enabling developers to add additional loads, power sources, or subsystems with minimal changes to the existing codebase. This flexibility ensures that the simulator can evolve alongside the increasing complexity of spacecraft missions and can accommodate new technologies or mission-specific requirements. Moreover, the integration with SIMULUS ensures compatibility with existing European Space Agency (ESA) simulation standards, facilitating interoperability and potential reuse across multiple projects.

Beyond its immediate technical achievements, this simulator represents a practical step towards the development of Digital Twins for spacecraft. By combining

physics-based models with real-time telemetry and historical data, future iterations of the simulator could provide predictive capabilities for on-orbit system health monitoring, mission planning, and anomaly detection. Such developments would enable operators to make informed decisions during critical mission phases, reducing risks and increasing the probability of mission success.

While the current work focuses on the EPS subsystem and software simulation, several opportunities for future improvements have been identified:

- **Expansion of subsystems:** Integration of additional satellite subsystems, such as thermal control, attitude control, or communication payloads, would enable more comprehensive mission simulations.
- **Real-time interaction:** Incorporating a real-time interface with actual hardware components or ground station simulators could enhance the fidelity of the simulation.
- **Advanced modeling:** Implementing more detailed battery degradation models, solar array aging, or fault injection scenarios would improve predictive capabilities and reliability assessment.
- **Digital Twin integration:** Linking the simulator with on-board telemetry and fleet history could support real-time monitoring and predictive maintenance strategies, effectively creating a full-scale Digital Twin.

In conclusion, this thesis has demonstrated the feasibility and benefits of developing a modular, scalable, and validated EPS simulator for satellite missions. The work provides a solid foundation for future research, testing, and operational support. By enabling engineers to simulate complex operational scenarios safely and efficiently, this simulator contributes to the advancement of satellite power system design and management, while also offering a pathway toward comprehensive Digital Twin implementations.

The successful completion of this project highlights the importance of simulation in modern space engineering, where cost, risk, and system complexity demand advanced virtual testing environments. The EPS simulator developed in this work exemplifies how rigorous engineering methods, combined with modular software design and adherence to industry standards, can produce reliable and extensible tools that enhance both the development process and mission safety.

Bibliography

[1] J. Gang C. Daehnick and I. Rozenkopf. *Space launch: Are we heading for oversupply or a shortfall?* Mckinsey & Company Aerospace & Defense. Apr. 2023. URL: https://www.mckinsey.com/industries/aerospace-and-defense/our-insights/space-launch-are-we-heading-for-oversupply-or-a-shortfall/ (cit. on p. 1).

[2] NASA Johnson Space Center. *Simulation & Modeling*. [Online; accessed 19-August-2025]. June 2025. URL: https://www.nasa.gov/reference/jsc-simulation-modeling/ (cit. on p. 2).

[3] NASA. *Enabling Simulation Interoperability between International Space Agencies*. [Online; accessed 20-August-2025]. July 2022. URL: https://ntrs.nasa.gov/api/citations/20220009047/downloads/DS_RT_2022.pdf (cit. on p. 2).

[4] E. H. Glaessgen and D. S. Stargel. *The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles*. [Online; accessed 21-August-2025]. American Institute of Aeronautics and Astronautics. Mar. 2012. URL: https://ntrs.nasa.gov/api/citations/20120008178/downloads/20120008178.pdf (cit. on p. 3).

[5] European Space Agency (ESA). *Introduction to SIMULUS*. [Online; accessed 27-August-2025]. 2025. URL: https://sim.space-codev.org/simulus/introduction/ (cit. on p. 8).

[6] Keen Lee. *Why Use Linux?* [Online; accessed 05-September-2025]. 2025. URL: https://www.suse.com/c/why-use-linux/ (cit. on p. 14).

[7] European Space Community. *System Design Overview*. Accessed: Sep. 10, 2025. 2025. URL: https://sim.space-codev.org/docs/10.5.0/simsat/sdd/system_design_overview (cit. on p. 15).

[8] Ferdinando Tonicello. *Electrical & Electronics Background Reading*. [Online; accessed 27-August-2025]. ESA Academy / ESTEC. Sept. 2021. URL: https://ecss.nl/wp-content/uploads/2021/09/STC_21-P-Electrical-and-Electronic_background_reading.pdf (cit. on pp. 20, 21).