aerospace

Article

A Digital Twin Approach for Spacecraft On-Board Software Development and Testing

Andrea Colagrossi *, Stefano Silvestrini *, Andrea Brandonisio and Michèle Lavagna

Department of Aerospace Science and Technology, Politecnico di Milano, Via La Masa 34, 20156 Milan, Italy; andrea.brandonisio@polimi.it (A.B.); michelle.lavagna@polimi.it (M.L.)
* Correspondence: andrea.colagrossi@polimi.it (A.C.); stefano.silvestrini@polimi.it (S.S.)

Abstract

The increasing complexity of spacecraft On-Board Software (OBSW) necessitates advanced development and testing methodologies to ensure reliability and robustness. This paper presents a digital twin approach for the development and testing of embedded spacecraft software. The proposed electronic digital twin enables high-fidelity hardware and software simulations of spacecraft subsystems, facilitating a comprehensive validation framework. Through real-time execution, the digital twin supports dynamical simulations with possibility of failure injections, enabling the observation of software behavior under various nominal or fault conditions. This capability allows for thorough debugging and verification of critical software components, including Finite State Machines (FSM), Guidance, Navigation, and Control (GNC) algorithms, and platform and mode management logic. By providing an interactive and iterative environment for software validation in nominal and contingency scenarios, the digital twin reduces the need for extensive Hardware-in-the-Loop (HIL) testing, accelerating the software development lifecycle while improving reliability. The paper discusses the architecture and implementation of the digital twin, along with case studies based on a modular OBSW architecture, demonstrating its effectiveness in identifying and resolving software anomalies. This approach offers a cost-effective and scalable solution for spacecraft software development, enhancing mission safety and performance.

Keywords: digital twin; spacecraft on-board software; processor-in-the-loop; hardware-in-the-loop; real-time simulation; finite state machine; GNC algorithms

1. Introduction

The rapid growth in the complexity of spacecraft On-Board Software (OBSW) is reshaping how design, verification, and validation (V&V) must be conducted. Modern missions increasingly depend on autonomy, reconfigurable avionics, and distributed architectures that must operate reliably under uncertain, time-varying conditions. As a result, software has become the core vehicle for mission safety and performance, elevating the assurance of correctness, robustness, timing determinism, and resource awareness to first-order concerns for space system engineers [1–4]. Traditional validation and verification approaches—unit testing, code reviews, and Hardware-in-the-Loop (HIL) test beds—remain indispensable, but they are often rigid, costly to evolve across design iterations, and insufficiently expressive to capture the tight coupling between embedded software, avionics interfaces, and closed-loop mission dynamics. They also tend to lack a modular and reusable testing philosophy that can accompany the system from early

Academic Editor: Marco Sabatini

Received: 1 December 2025
Revised: 23 December 2025
Accepted: 24 December 2025
Published: 6 January 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

 https://doi.org/10.3390/aerospace13010055

prototyping to flight acceptance, and they struggle to exercise contingency scenarios in which faults and anomalies must be injected at the protocol, electrical, and software layers, including hybrid hardware–software configurations.

These limitations motivate a shift toward development and test frameworks that are (i) composable and reconfigurable, (ii) lifecycle-aware and CI/CD-friendly, and (iii) capable of end-to-end, dynamics-in-the-loop assessment with systematic fault injection and coverage traceability. To address these challenges, the concept of the *digital twin* has gained prominence across the aerospace sector. A digital twin represents a high-fidelity, dynamic digital counterpart of a physical asset, designed to mirror its configuration, state, and behavior in real time. Initially developed for industrial and manufacturing systems, digital twin approaches have recently been applied to space systems engineering to support system monitoring, predictive maintenance, and software validation [5,6]. In this context, the digital twin functions as an executable model that couples the spacecraft's physical dynamics with its software logic, enabling co-simulation and continuous verification through virtualized execution environments.

1.1. Related Work

Several recent studies have demonstrated the relevance of digital twins for spacecraft and space robotics. He et al. [5] proposed a digital twin-based networking solution for low-Earth orbit constellations, highlighting the potential for autonomous fault recovery and system adaptation. Chiti et al. [6] extended the concept to quantum satellite networks, illustrating how virtualized twins can assist software-defined communication architectures. In the field of planetary exploration, Pinello et al. [7] developed a digital twin of a rover to detect structural damage using sensor data fusion, demonstrating the benefit of dynamic modeling for system resilience. Moreover, Hou et al. [8] introduced a digital twin-based framework for runtime verification and cybersecurity protection of satellite systems, suggesting a broader role for digital twins as safety monitors during mission operations. NASA's Orion program has similarly adopted SysML-based modeling and digital twin integration for the Artemis I mission, providing lessons on traceability and testing of complex flight software [9]. Verification and validation of spacecraft on-board software have traditionally relied on a combination of Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL), and Hardware-in-the-Loop (HIL) testing approaches. These methodologies are well established across both large-scale and small satellite programs and remain essential for final system qualification. Several works have demonstrated the effectiveness of HIL environments in exposing integration issues, timing anomalies, and interface-related faults that are difficult to detect through purely software-based testing. In particular, authors in [10–12] presented a HIL environment for the verification of small satellite on-board software, highlighting the benefits of using flight-representative interfaces and processors. More recently, Köstler et al. [11] described a combined SIL/HIL campaign for the MOVE-II CubeSat, demonstrating how early-stage software testing can significantly reduce integration risks. Despite their effectiveness, HIL facilities are often costly, rigid, and difficult to adapt across design iterations. Reconfiguration of hardware-centric testbeds typically requires extensive manual intervention, physical rewiring, and repeated validation procedures, which limits their suitability for early and continuous software development. These challenges have motivated the adoption of software-driven and processor-based testing environments aimed at shifting verification activities toward earlier phases of the development lifecycle. Di Capua et al. [13] proposed a software-based environment that integrates processor emulation, spacecraft dynamics, and fault injection to validate OBSW logic before hardware availability, demonstrating reduced iteration time and improved fault coverage. Along similar lines, El Wafi et al. [14] introduced a processor-in-the-loop framework for

CubeSat ADCS development, emphasizing the role of real-time execution in validating control and monitoring software. Parallel to these developments, attitude control and GNC-specific HIL facilities have been extensively investigated. Nicolai et al. [12] presented recent advancements in HIL testing for spacecraft attitude control systems, underscoring the importance of real-time closed-loop validation when testing guidance and control algorithms. Other processor-based and FlatSat-oriented testbeds have also been reported to support real-time verification of nanosatellite ADCS and GNC functions, particularly in academic and research laboratory settings [15,16]. More recently, the concept of the digital twin has emerged as a unifying paradigm capable of bridging high-fidelity simulation, embedded software execution, and lifecycle support. Glaessgen and Stargel [17] provided one of the earliest aerospace-oriented definitions of the digital twin, framing it as an ultra-high-fidelity virtual counterpart that accompanies the physical system throughout its lifecycle.

Nevertheless, most existing digital twin implementations in the space sector focus primarily on system-level modeling, communication architectures, or post-launch monitoring. Comparatively fewer works address the application of digital twins to real-time embedded software testing, particularly for OBSW functions such as finite state machines, fault detection, isolation and recovery logic, and platform management under strict timing constraints. Comprehensive verification strategies for onboard control software have been proposed at a methodological level [18], yet their integration with real-time, flight-representative execution environments remains limited.

1.2. Contributions

These works collectively underscore the emerging role of digital twins as an enabling technology for the next generation of spacecraft software development and validation pipelines. However, most existing implementations focus primarily on system-level or communication aspects, while the use of digital twins for embedded software testing—especially for real-time OBSW—remains limited. Current practices often separate hardware and software validation stages, leading to fragmented workflows and delayed feedback cycles. There is thus a growing need for frameworks that unify high-fidelity hardware emulation, software execution, and fault-injection capabilities in an integrated environment.

This paper addresses this gap by presenting a *digital twin* for spacecraft OBSW development and verification. The proposed approach enables real-time execution of embedded software in a simulated environment, supporting the observation and analysis of system behavior under both nominal and fault conditions. It provides a testing framework for critical functions such as Finite State Machines (FSMs), Guidance, Navigation, and Control (GNC) algorithms, and platform management logic. By offering an interactive and iterative validation process, tested within the ASTRA Lab at Politecnico di Milano, the digital twin reduces reliance on costly HIL setups, accelerates the software development cycle, and improves the overall reliability of mission-critical software.

Beyond functional software validation, one of the primary challenges in mission development lies in reducing the time, cost, and rigidity associated with traditional hardware-centric testing campaigns, while preserving their essential role in final qualification. In this context, digital twin-based approaches enable a shift of a significant fraction of verification activities toward earlier, software-driven development stages.

In summary, the main contributions and benefits of this work are:

- **Digital twin framework for OBSW development and testing:** The paper introduces a modular digital twin architecture that couples real-time high-fidelity simulation with processor-in-the-loop execution of flight-representative on-board software.

- **Reduction of testing time and development iterations:** By enabling early-stage validation, rapid reconfiguration, and software-based fault injection, the proposed approach significantly reduces the time required to set up and re-run test campaigns compared to traditional Hardware-in-the-Loop facilities.
- **Improved fault detection and recovery validation:** The framework allows systematic and repeatable injection of sensor, actuator, protocol, and timing faults, improving the observability and verification of fault-detection, isolation, and recovery (FDIR) logic.
- **Enhanced software reliability prior to HIL testing:** The digital twin increases software maturity before final qualification, reducing the number of costly HIL iterations while preserving HIL testing for final acceptance.
- **Lifecycle continuity from development to operations:** The same digital twin facility can be reused for post-launch anomaly reproduction and validation of corrective actions, extending its benefits beyond ground testing.

The remainder of this paper is organized as follows. Section 2 describes the overall digital twin framework; Section 3 presents the high-level software development, including example implementations of the proposed mode management architecture; Sections 4 and 5 describe results for continous development and testing made possible by the digital twin; finally, Section 6 draws conclusions for the work and delineates future direction for spacecraft digital twin development.

2. The Digital Twin Framework

The proposed digital twin framework for spacecraft On-Board Software (OBSW) development and testing is structured around three main pillars: a hardware-in-the-loop inspired flat-sat facility, a high-fidelity software simulation environment and a formal development tool for OBSW logic. Together, these elements enable real-time execution of spacecraft subsystems, ensuring a seamless integration of flight processors with virtualized sensors, actuators, and dynamic environments while guaranteeing coherent FSM definition and logic validation. The framework provides a modular and scalable architecture, supporting both subsystem-level verification and system-level validation, while reducing reliance on costly physical testbeds (see Figure 1).

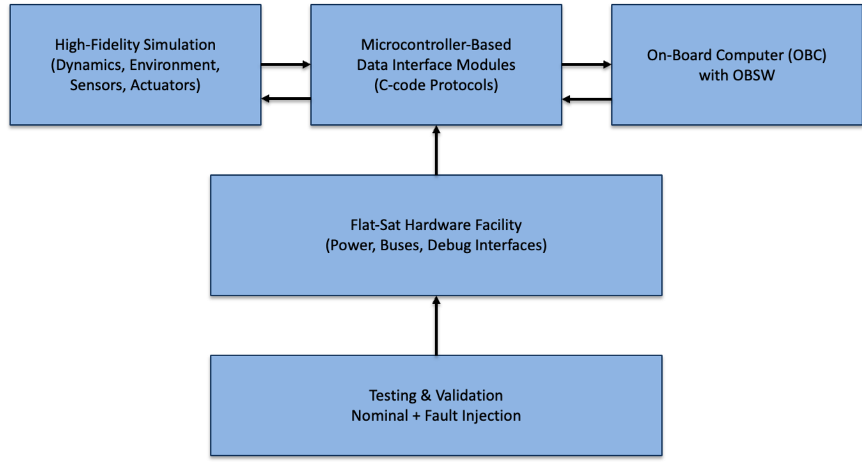

Figure 1. Overview of the digital twin framework for spacecraft On-Board Software development and testing.

The scalability of the digital twin framework is inherently guaranteed by its modularity: each additional subsystem, redundant unit, or distributed on-board computer can be integrated by instantiating further microcontroller-based interfaces and extending the simulation models accordingly. For larger spacecraft with distributed architectures

and higher redundancy levels, the same principles apply, with the digital twin naturally evolving toward a network of synchronized virtual nodes. The primary scalability limit is not conceptual but computational, and can be addressed by distributing simulation workloads across multiple real-time targets with shared time reference.

2.1. Hardware

The hardware branch of the digital twin is realized through a modular flat-sat facility designed to electrically and functionally replicate the spacecraft bus and its interfaces. Its architecture emphasizes modularity, safety, and fidelity to flight conditions, while preserving accessibility for debugging and incremental integration.

The flat-sat consists of a set of PC104-compatible breakout boards that expose the electrical connections of the On-Board Computer (OBC) and avionics modules, providing testpoints and debug lines for external instrumentation. Dedicated power distribution boards ensure redundant and safe delivery of power to the components under test, with the capability of controlled hot-swapping during debugging sessions. A core interface board unpacks the OBC signals, exposing all relevant spacecraft communication buses (e.g., CAN, I^2C, SPI, UART, LVDS) and allowing termination at configurable debug interfaces. This design facilitates fast identification of anomalies and robust regression testing across the OBSW development cycle.

A critical feature of the facility is its capability to host electrically isolated digital twins of spacecraft sensors and actuators. Data interface modules, based on programmable micro-controllers, reproduce in detail the electrical protocols, timing, and functional behaviors of the real hardware. Classical interfaces supported include UART (RS-232/RS-422), I^2C, SPI, CAN, as well as analog and PWM signals. The use of digital isolators guarantees the electrical safety of the OBC and prevents spurious couplings, while maintaining high-speed, loss-free data transmission. This modular architecture allows individual subsystems—such as power management or solar array electronics—to be virtualized and seamlessly integrated into the OBC's operational loop (see Figure 2).

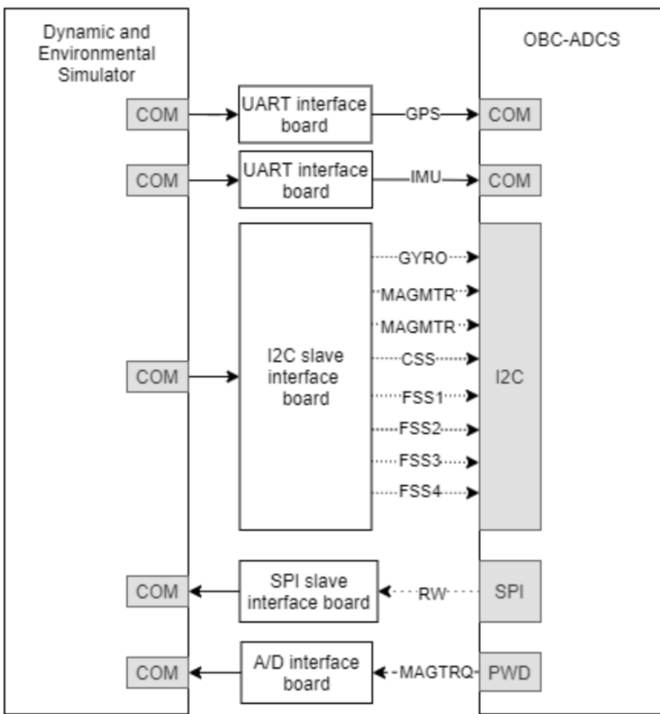

Figure 2. Hardware and software interface structure between the On-Board Computer, microcontroller-based data interface modules, and the high-fidelity simulation environment.

Through this setup, the digital twin enables closed-loop, real-time processor-in-the-loop (PIL) testing without relying on complex moving platforms such as frictionless bearings or Helmholtz cages. When required, hybrid configurations can be implemented by substituting specific virtualized peripherals with their real counterparts, thus bridging between processor-in-the-loop and hardware-in-the-loop campaigns. This flexibility supports both early-stage functional debugging and high-fidelity system validation in later development phases. A picture of the digital twin facility in its flat-sat configuration is shown in Figure 3.

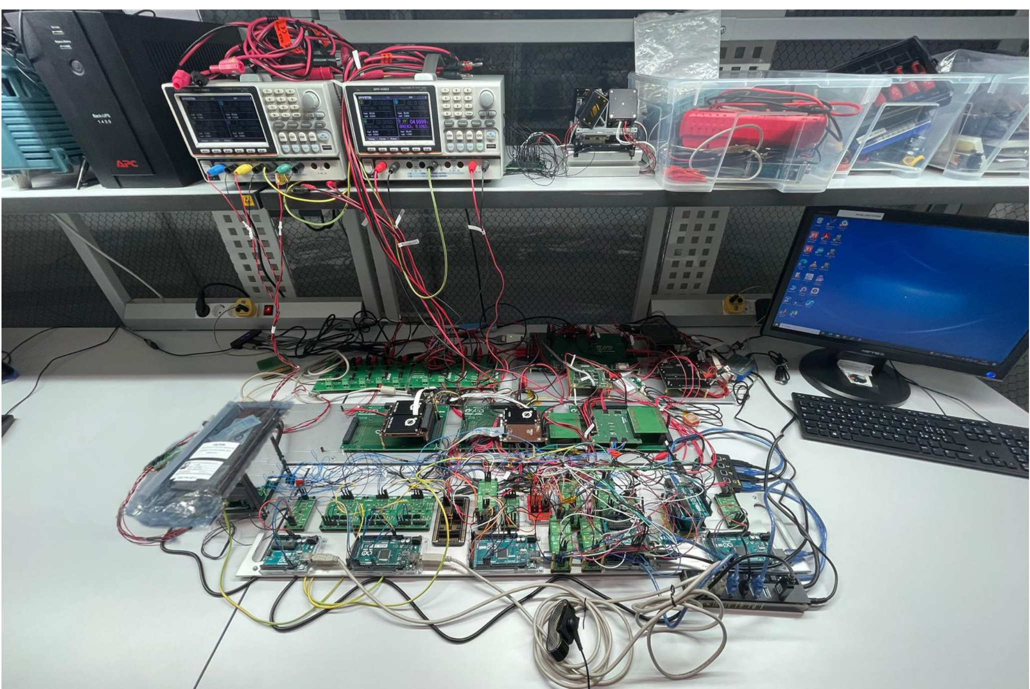

Figure 3. Representative set-up of a digital twin in a complete flat-sat configuration, with laboratory power supplies used to reproduce the solar power generation of solar arrays from a simulated environmental scenario.

2.2. Simulation Software

The software branch of the framework encompasses both the real-time simulation environment and the embedded software defining the digital twin of hardware peripherals. At its core lies a validated Functional Engineering Simulator (FES) developed in MATLAB/Simulink, which incorporates a Dynamics–Kinematics–Environment (DKE) module consistent with ECSS standards. This simulator provides high-fidelity models of spacecraft sensors, actuators, and orbital environment, ensuring numerical equivalence with real hardware outputs in terms of format, frequency, and datatype. The Simulink Desktop Real-Time (SDRT) kernel is employed to guarantee deterministic execution on standard workstations, achieving update rates up to 20 kHz and allowing strict monitoring of real-time performance.

Real-time determinism and synchronization are ensured by a master–slave timing architecture. The real-time simulator acts as the global time reference, executing with a fixed step under the SDRT kernel. Sensor and actuator interface microcontrollers operate synchronously with the simulator, exchanging time stamped data at predefined rates and respecting strict communication deadlines. Fault injection does not alter the global time synchronization base but is implemented by modifying data validity, timing, or protocol behavior at the interface level. This guarantees that even under injected anomalies, the sys-

tem preserves deterministic execution and coherent time alignment between simulation, interfaces, and on-board software.

Sensor and actuator data generated in the simulator are routed through the data interface modules, where embedded software written in C runs on microcontrollers. These microcontrollers handle the detailed definition of the communication protocols, faithfully reproducing the timing, commands, and responses of the real hardware. Each microcontroller can represent a single sensor/actuator or a group of peripherals connected to a common bus. In this way, the OBSW interacts with the digital twin through the exact same electrical and logical interfaces as in flight, ensuring that bus contention, synchronization, and protocol handling are rigorously verified.

The combination of high-fidelity simulation and protocol-level emulation enables the observation of OBSW behavior under a wide variety of conditions, including nominal, degraded, and failure-injected scenarios. This setup allows GNC algorithms, power and data handling logic, and platform mode management software to be iteratively tested and refined before advancing to hardware-in-the-loop or integrated system campaigns.

In summary, the software component of the digital twin provides both the execution environment for virtual subsystems and the embedded software defining their interfaces. By combining real-time physical modeling with microcontroller-based communication emulation, the framework establishes a powerful environment for early validation, fast iteration, and robust verification of spacecraft embedded software.

2.3. Finite State Machine Formal Verification

The digital twin framework is completed by a formal verification tool to develop and assess the logical structure of the OBSW. Specifically, the development and verification of FSM and FDIR routines is based on a virtual model of the software architecture implemented in the OpenGEODE (https://essr.esa.int/project/opengeode, accessed on 15 October 2025) environment using the SDL formalism. In this context, the SDL-based model does not merely provide a specification of the FSM; rather, it acts as a digital twin of the on-board software logic, enabling the systematic design, verification, and refinement of the finite state machine in a controlled environment before deployment on the spacecraft. The use of SDL is particularly suitable for digital twin implementations since the language is formally complete and directly supports both simulation and automatic code generation. Its modelling paradigm is based on four principal constructs: structure, communication, behaviour, and data.

- **Structure**: The digital twin is hierarchically organised into function blocks, which can be decomposed into sub-blocks down to the process level. Processes describe the detailed operations of the FSM and map directly to executable states in the real system.
- **Communication**: Blocks exchange information via communication channels, which in the digital twin can replicate both internal (intra-system) and external (ground-to-spacecraft) signal flows, thus capturing the dynamics of nominal and contingency scenarios.
- **Behaviour**: Each process defines the complete set of actions available to the FSM, such as reading inputs, sending messages, cycling through tasks, making decisions, or calling procedures. This provides the backbone for scenario-based simulations of the on-board decision logic.
- **Data**: All variables used within the processes are defined according to a formal syntax, supporting primitive data types (e.g., integers, floats, booleans, strings). This guarantees consistency between the digital twin and the eventual embedded implementation.

By formalising the FSM through SDL and implementing it in OpenGEODE, the digital twin provides a reliable tool for virtual prototyping, formal verification, and early validation of the on-board software. The approach is especially well-suited for nano-satellites'

missions, where simplicity and robustness are paramount, and where testing opportunities are often limited due to resource constraints. The digital twin enables iterative evaluation of monitoring parameters, failure responses, and transition logic across the different software modes, ensuring that design errors are detected and mitigated before flight implementation.

An example of OpenGEODE formalism is shown in Figure 4 concerning the SW-MAIN INIT mode, presented in Section 3.1.1. This digital twin implementation not only validates the logical architecture but, eventually, may also provide a direct foundation for code generation, thereby facilitating the translation of the SDL representation into the flight software deployed on the spacecraft's OBC.

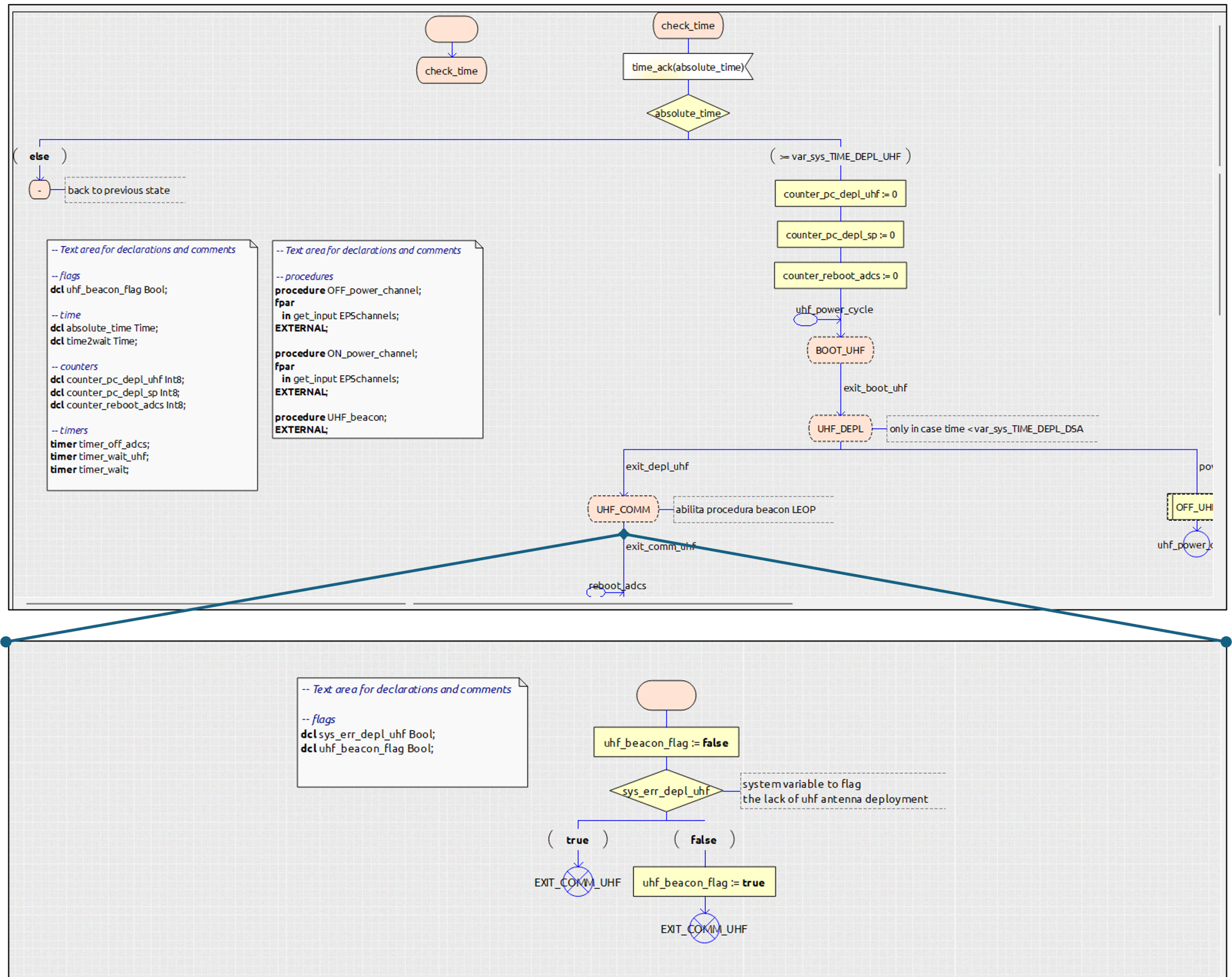

Figure 4. OpenGEODE snapshot of the initial implementation of the SW-MAIN INIT mode, with an inside look at a specific UHF-COMM state application.

3. Spacecraft On-Board Software Development

Starting from missions' typical high-level functionalities, some FSM macro-modes, such as INIT, including all the required early operations procedures, nominal routine (NOM) and safe mode (HOLD), are identified, and the connecting transitions among them are defined. Then, the logic underlying each mode is described, accounting for either the relevant requirements to be satisfied, the mission's operations and the actual functioning of the spacecraft's components. Secondly, to verify compatibility with the spacecraft's design and operations, the logic's design is supported and consolidated through analyses involving the spacecraft subsystems, such as the attitude determination and control

(ADCS), power generation and management, and on-board data handling. During such a process, particular attention is paid to the design of the nano-satellite's FDIR and to its translation into the FSM logic, which shall be achieved by means of monitoring functions that perform the Fault Detection (FD) and the safe routines, which are devoted to the Fault Identification (FI) and, possibly, fault correction. Afterwards, the derived FSM logic is represented in detail with the Specification and Description Language (SDL), which allows the formal verification of the FSM and bridges the high-level definition to the final on-board software implementation.

Despite mission-specific objectives, a large number of functionalities need to be always granted by a general nano-satellite's on-board software. These include not only common low-level tasks, such as the boot and start-up of the system, the communication with peripherals and the monitoring of on-board components, but most importantly high-level functionalities. The latter range from on-board tasks as communication with ground, power generation, distribution and storage, and (for the more complex nano-satellites' missions) the attitude determination and control of the spacecraft; up to operational functionalities, related to the ground planning and operation of the satellites, which are tightly related to the mission phases.

3.1. High-Level Design

In this section, a generally applicable backbone structure for the on-board software of nano-satellites is proposed and discussed. The design follows an FSM-based approach, in which the software backbone is defined directly in terms of the software modes.

The key insight behind the definition of the FSM modes lies in recognizing the distinct underlying working principles required to accomplish different high-level software functionalities. In other words, the FSM modes are derived by grouping together some functionalities that can be implemented using the same set of drivers. It is important to emphasize that this identification process focuses exclusively on high-level software functions, since low-level functionalities are assumed to be continuously active throughout the entire operation of the software.

For a small-sized spacecraft software, three main high-level functionality groups—based on similar working principles—are identified:

1. Tasks that must be executed only once during the mission, typically at the beginning of space operations, for which full satellite autonomy is essential, since the system is not yet capable of communicating with the ground.
2. Tasks that require repeated execution, where ground control and planning are preferable.
3. Tasks that may also be executed repeatedly during the mission, but whose occurrence depends on contingencies and, therefore, cannot be planned in advance or addressed immediately by the ground mission control.

The above-mentioned high-level clustering can be mapped into three fundamental software macro-modes, which form the basis of the nano-satellite software structure. Specifically, these modes are referred to as **INIT** (Initialization), **NOM** (Nominal) and **HOLD** (hold and safe mode).

1. The INIT mode covers all the operations that start with the spacecraft's first switch-on in space, and must be executed only once (e.g., system boot, appendages deployment and initial detumbling). The driving principle here is the one-time, successful execution of a sequence of tasks. Moreover, since batteries are often the primary power source for these types of missions, such autonomous tasks must be completed within a limited timeframe. It is therefore proposed to base this mode on time-tagged commands, which are repeatedly executed until either the success of each task is con-

firmed or the allocated timeout expires. Such a software mode should be employed during the first part of the LEOP (Launch and Early Operations Phase) mission phase.
2. The NOM mode relies on regular communication with the ground and is typically employed during spacecraft commissioning as well as throughout all nominal mission phases. In this mode, operations are based on scheduled commands uploaded from the ground during communication windows. Each command is associated with a scheduled execution time, at which the on-board software triggers its operation.
3. The HOLD mode consists of automatic routines triggered when an error is detected and cannot be resolved, or more generally, when immediate ground contact would be required but is not available during the occurrence. Therefore, the driving principle of HOLD mode is to maximize spacecraft survivability by increasing the probability of establishing communication with the ground and maintaining a positive power balance.

Each of the three major software modes requires both entry and exit transitions, which can generally be automated or commanded by ground.

1. INIT mode is primarily defined by its time-triggered nature. Entry occurs when the elapsed time since the first switch-on of the on-board computer in space is below a predefined threshold. This threshold should be established and validated through analyses and tests. Then, INIT mode is exited once this time threshold is exceeded, triggering a transition to HOLD mode. This transition is motivated by the fact that HOLD ensures a positive power balance while maintaining basic system functionalities and maximizing the chances of achieving first ground contact.
2. NOM mode can only be entered through a ground command, following a successful communication contact established in HOLD mode and requiring the upload of a command schedule. Exit from NOM mode, on the other hand, is driven by contingencies—i.e., failures or non-nominal conditions detected on board—which trigger an automatic transition to HOLD mode.
3. HOLD mode can be entered either from INIT or NOM in response to on-board events. Exit from HOLD is possible only through ground intervention, which commands the transition to NOM mode.

In addition to the macro-modes defined above, to ensure that all possible anomalies on board can be addressed, it is proposed to integrate each FDIR function into the software FSM through dedicated monitors and routines, as outlined below:

- **FD (Fault Detection)**. The integrated monitors continuously check the system variables required for the correct operation of the platform. These monitors should be tailored to the specific FSM mode in use.
- **FI (Fault Isolation)**. When a monitor detects a failure, dedicated routines are responsible for fault isolation. The most severe outcome of such a procedure is a request to transition the entire system into HOLD mode, where nominal platform functions are halted to establish fast and safe communication with the ground.
- **FR (Fault Recovery)**. Recovery functions are embedded within the previous FI routines. Once an anomaly has been isolated, the software may attempt autonomous corrective actions, i.e., rebooting the component related to the failure. Only a limited set of autonomous recovery actions should be implemented on board, specifically those required for immediate intervention to prevent catastrophic system loss. In all other cases, recovery should involve ground control. For this purpose, the system remains in HOLD mode while awaiting ground commands.

It is proposed that FDIR monitors and routines operate in all system modes, with the sole exception of INIT. This is because the INIT phase is inherently designed to execute

tasks, verify their accomplishment, and repeatedly attempt them until success is achieved. During NOM mode, parameter monitors and FI-FR routines run in parallel with the scheduled tasks. Furthermore, in HOLD mode, the monitors and FI-FR routines remain active. Indeed, the proposed approach offers the advantage of enabling autonomous fault isolation and recovery attempts, under specific conditions, before interrupting current tasks and awaiting ground intervention. In this way, prompt recovery can be attempted immediately at the time of failure, while simple fault cases may be resolved on board, ultimately reducing mission downtime and improving operational continuity.

The main challenge of nano-satellite missions lies in achieving mission objectives with a low-cost design, which inherently involves accepting higher levels of risk. Compared to larger-scale missions, both the probability and severity of failures are often greater, due to factors such as the absence of redundancies, tighter design margins in power and telemetry budgets, and the typically reduced scope of testing campaigns for small spacecraft. Therefore, a key advantage of adopting the proposed macro-modes is the ability to implement a modular and linear FSM. Unlike hierarchical structures, this approach is more easily portable across different missions and can be readily adapted to specific requirements. Nevertheless, mission-specific factors strongly influence the design output. However, the approach to be adopted during the detailed design phase can still be standardized by identifying and exposing the major software parameters of interest.

The process should begin with a thorough understanding of the functionality of the nano-satellite's COTS components, to determine which parameters can be monitored and which commands can be issued. Any missing or custom software elements should be identified and addressed in the earliest design stages. In addition to designing, implementing, and testing low-level software components, the FSM routines should be verified through a multidisciplinary approach—ensuring a holistic perspective on the mission software while also accounting for subsystem-specific requirements. Detailed design should follow system engineering principles and proceed iteratively, so that mission-specific components, tasks, and operations can be progressively incorporated. It is also recommended to conduct multidisciplinary analyses across different subsystems to validate the expected software behaviour. In particular, simulations and analyses should include attitude determination and control, power budgets, and housekeeping telemetry budgets. Some key FSM-related trade-offs that typically involve multiple subsystems include:

- Defining the priorities and timing of the INIT mode tasks sequence;
- Selecting a restricted set of parameters to monitor and defining the related FI-FR routines;
- Determining which events to log and assessing their severity.

The output of the detailed FSM design shall be the clear schematics of the different modes, comprehensive parameter monitoring lists, and schematics of the FI-FR routines.

In general, for nano-satellite design, the OBC board is decoupled from the ADCS board. Therefore, it is essential to define and fix the interface between the two. The main onboard computer serves as the central interface for all spacecraft components, including the scientific payload, power and telecommunication modules, and the ADCS board itself. In contrast, the ADCS system interfaces with all attitude determination sensors and control actuators. In the following, the two onboard computers will be referred to as OBC-MAIN and OBC-ADCS. Following the proposed methodology, the OBC software high-level structure is divided into three main modes—INIT, NOM, and HOLD—corresponding to the three primary functionality principles for nano-satellite software. The INIT mode boots the system, the ADCS, and the main communication system, enabling rapid establishment of the first ground contact. Additionally, it handles the deployment of folded appendages (e.g., communication antennas and solar arrays) and performs satellite detumbling. The NOM mode is used for routine operations: executing attitude manoeuvres,

commanding the scientific payload, maintaining communication with ground, and monitoring the critical satellite parameters. Finally, the HOLD mode focuses on minimizing power consumption, establishing fast communication with the ground, controlling the ADCS to maximize power generation, and handling non-nominal situations such as tumbling or unexpected wheel saturation. The software running on each OBC—referred to as SW-MAIN and SW-ADCS—is structured according to its own FSM, reflecting the functionality principles described above. The two systems are interfaced through structured data commands and monitoring parameters within a master-slave logical architecture. Specifically, SW-ADCS is subordinated to the commands and requests defined by SW-MAIN. This high-level structure is illustrated in Figure 5. Both FSMs follow the same architecture defined by the three main modes, with the exception that SW-ADCS does not implement INIT, as it is unnecessary given its subordination to SW-MAIN. The master-slave relationship is justified by the fact that SW-ADCS exchanges monitoring data with SW-MAIN while only receiving commands, ensuring coherent operation across both systems.

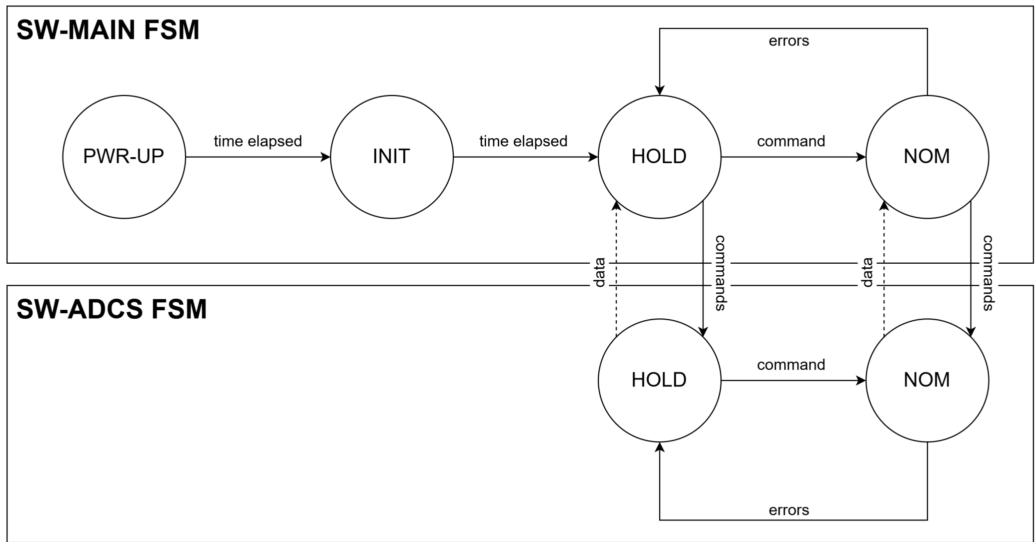

Figure 5. OBCs software high-level structure.

The transitions between modes, along with their respective entry and exit conditions, are defined in accordance with the general-purpose methodology. HOLD serves as the INIT exit condition, as it is designed to maintain a positive power balance and seek ground contact, while the first schedule upload triggers the transition to NOM mode. Both software systems incorporate FI-FR routines, which continuously monitor vital parameters to detect potential failures or errors—SW-MAIN for the general system and SW-ADCS for the ADCS subsystem.

In the following sections, each software mode will be analyzed in more detail, distinguishing between the two software systems: SW-MAIN and SW-ADCS.

3.1.1. MAIN

As shown in Figure 5, the FSM of the SW-MAIN is divided into three main modes: INIT, NOM and HOLD. The design of each mode is hereby briefly described.

INIT Mode

In SW-MAIN INIT mode, every operation is time-tagged. Nominally, the correct operational sequence is illustrated in Figure 6 and described in the following list. Since the process is time-tagged, all operations are confined to specific time slots, within which the software cyclically attempts to execute each task, as shown again in Figure 6.

1. The OBC-MAIN board powers on and performs its automatic boot sequence.
2. Once the OBC is operational, it waits until the end of the short *slot 1*, then boots the main communication link, and deploys its antennas, if any. If both the boot and deployment are successful, the COMM system is commissioned, and the board begins transmitting beacon messages to establish first ground contact.
3. After COMM commissioning, the INIT mode boots the OBC-ADCS board.
4. If the ADCS boot is successful, the software commands the ADCS to activate detumbling mode to reduce angular velocity before attempting deployment of the solar arrays, which cannot withstand high rotational rates.
5. At the conclusion of *slot 2*, the software commands the deployment of the spacecraft's solar arrays, even if detumbling is not yet fully complete. This is because the risk of deploying solar panels at a high tumbling rate is considered lower than the risk of depleting the batteries while keeping the panels folded during prolonged detumbling.

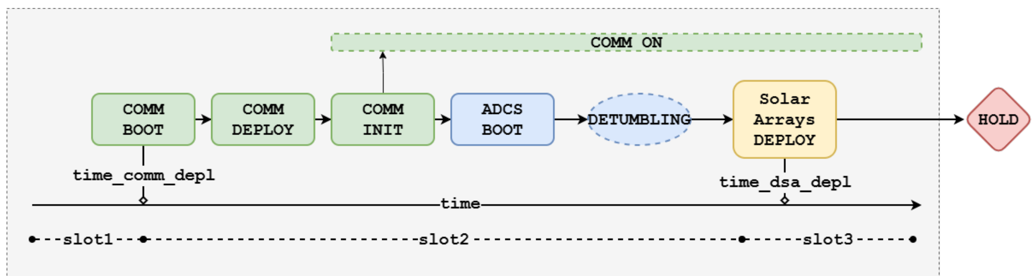

Figure 6. SW-MAIN INIT mode operational schematic.

Upon completion of this process, the OBC system autonomously transitions to HOLD mode. This INIT mode is intentionally structured to ensure the prompt and reliable deployment of the solar panels, as their availability is a mission-critical requirement for spacecraft survival. Failure to deploy the panels would severely compromise platform operability and result in mission loss. This consideration emphasizes the necessity of adopting a time-tagged rather than event-tagged design for the INIT phase, thereby guaranteeing solar panel deployment at a predetermined time, even if preceding operations have failed.

NOM Mode

The SW-MAIN nominal mode is based on the considerations introduced in Section 3.1. In this configuration, the software executes commands defined within a schedule that is periodically uploaded from the ground during satellite operations. These scheduled commands are not limited to single actions but are instead implemented as Scripted Commands. Two categories of Scripted Commands are defined: Routine Scripted Commands, primarily employed during nominal operations, and Commissioning (or Contingency) Scripted Commands, used exclusively during the spacecraft commissioning phase or during contingency operations. The principal classes of Routine Scripted Commands are the following:

- **Manoeuvre commands**: Employed to request attitude manoeuvres from the ADCS. Inputs to this script include the guiding polynomial and the preferred actuator.
- **Observation commands** (start, make, stop): Used to activate and terminate scientific observation mode, employing the scientific payload in conjunction with the Iridium system as a low-latency communication channel.
- **Data transfer commands**: Used to transfer data acquired by the scientific payload into the OBC-MAIN internal memory.
- **Communication window commands** (start, make, stop): Employed to manage communication opportunities with the ground. Input arguments include the selected

communication channels and the targeted ground stations, nominally via the S-band telecommunication system.

By contrast, the Commissioning/Contingency Scripted Commands executed in NOM mode include activities related to the commissioning of the ADCS, TT&C, and EPS subsystems, as well as the payload, or to counteract the possible anomalies that may occur to one or more subsystems or components (i.e., missed deployment, error in the communication link, temperature range exceed, etc.).

Scripted Commands—both routine and commissioning—can be uploaded and modified from the ground, thereby ensuring operational flexibility. In addition, the schedule may include basic low-level actions and functions to be executed as required. As previously discussed, while Scripted Commands are executed in NOM mode, monitors and FI-FR routines remain active and periodically executed, thereby ensuring the FDIR capabilities of SW-MAIN and, by extension, of the entire satellite's system.

HOLD Mode

As introduced earlier, certain nominal monitors may trigger a transition into HOLD mode due to component errors or failures. The logical architecture of the HOLD mode follows a tree structure, in which system checks define branching conditions that ultimately lead to specific actions. Two main parameters govern the functioning of HOLD: the battery voltage and the ADCS status. Battery voltage is critical for assessing the remaining power available on the platform, while the ADCS status provides outputs in terms of either direct SW-ADCS requests or subsystem availability, thereby determining the satellite's operating mode. The logical scheme of HOLD is illustrated in Figure 7.

Upon entry into HOLD, the on-board computer set the communication systems in the default configuration, places the ADCS in standby, and sets the payload to power-save mode. In the event of errors originating from either the ADCS or the payload, their shutdown is enforced through the invocation of dedicated procedures. Following these initialization operations, the process enters the main central loop, which continuously monitors battery voltage and ADCS status. Exit from this loop is only possible through the reception of a ground command.

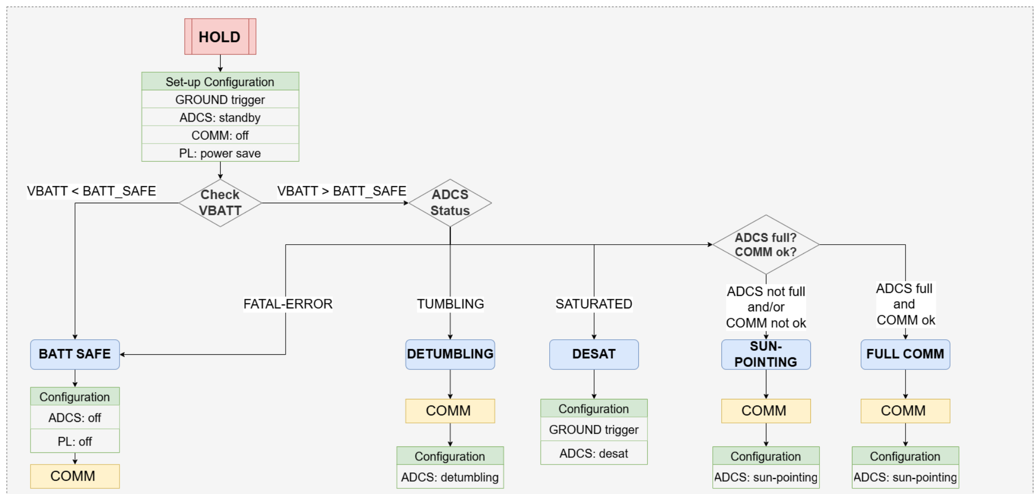

Figure 7. SW-MAIN HOLD mode operational schematic.

Depending on the conditions of the battery and ADCS, the process may follow one of five distinct branches (as shown in Figure 7): fatal error, detumbling, desaturation, safe ADCS, or nominal ADCS. A low battery voltage can directly trigger the fatal error branch if it falls below a predefined threshold. In such a critical condition, no operations are executed,

as the system will automatically shut down. In all other cases, the primary function of the FSM is to select the most suitable communication channel available and to establish ground contact as rapidly as possible.

3.1.2. ADCS

As shown in Figure 8, the FSM of the SW-ADCS is divided into two main modes: NOM and HOLD. The design of each mode is hereby briefly described.

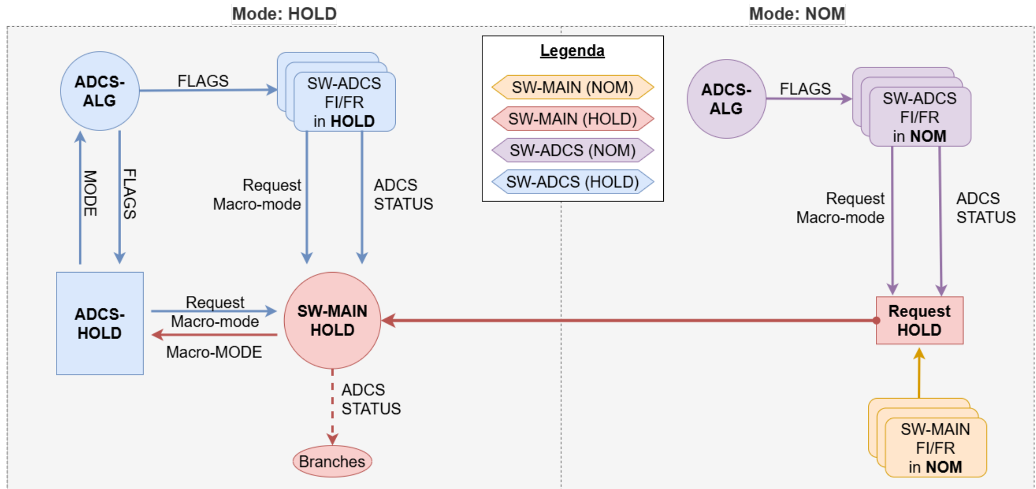

Figure 8. SW-ADCS NOM and HOLD modes operational schemes.

NOM Mode

The SW-ADCS NOM mode governs the nominal interaction between the ADCS algorithms and the associated monitors and FI-FR routines of the ADCS COTS components. The software architecture is organised into two primary elements: the ADCS algorithm and the ADCS monitors. The ADCS algorithm is responsible for guidance, navigation, and control (GNC) functions, and its operative mode is directly commanded by the SW-MAIN schedule. In parallel, the ADCS continuously monitors and evaluates the algorithm-generated flags to ensure the safe operation of the ADCS software, sensors, and actuators.

In the event of an error or failure, if the corresponding FI-FR procedure is unable to resolve the anomaly, the fault is escalated to SW-MAIN, which triggers a transition to the HOLD mode. This operational structure reflects the *master–slave* interaction described in Section 3.1, where SW-ADCS is subordinated to the control of SW-MAIN. The logical architecture of the SW-ADCS NOM mode is illustrated in the right-hand side of Figure 8.

HOLD Mode

The SW-ADCS HOLD mode maintains the coupling between the ADCS algorithms, monitors, and FI-FR routines, while introducing an additional element that becomes active exclusively in this mode: the ADCS-HOLD logic. This component is responsible for selecting the appropriate ADCS operative mode in accordance with the requests and instructions issued by the SW-MAIN HOLD tree logic. Once the SW-MAIN system transitions to HOLD mode—regardless of whether the transition was triggered by an ADCS-related anomaly—the SW-ADCS likewise enters its corresponding HOLD mode. In this configuration, the software is structured around three main actors: the ADCS algorithm, the ADCS monitors, and the ADCS-HOLD logic, as depicted on the left-hand side of Figure 8.

The ADCS algorithm retains its role from the NOM phase, continuing to process mode commands and expose status flags. The monitors, although slightly modified in HOLD mode, still provide information on ADCS status and mode availability, while also requesting macro-modes (defined as sets of operative modes characterised by common

subsets of sensors and actuators). The newly introduced ADCS-HOLD logic acts as the interface between the SW-MAIN HOLD procedure and the ADCS algorithm. Specifically, it processes macro-mode requests received from the monitors and, based on the availability of the requested macro-modes, selects and commands the first compatible operative mode to the ADCS algorithm.

3.2. MAIN Mode Manager Implementation

The ModeManager is a software component that implements a cascading FSM whose overall objective is the control of the operating steps of the satellite. The ModeManager receives three inputs and provides two outputs (see Figure 9).

Figure 9. Black-box view of the ModeManager component.

- **Inputs—**The timerEv signal is an n-bit vector where n is the number of timers in the computing platform of the satellite affecting the operations of the ModeManager. Each bit of the timerEv vector is associated with a specific timer, and when the bit is set signals the ModeManager that the corresponding timer has expired. Notably, the actual implementation of the ModeManager features a single timer due to the resource limitations of the employed hardware platform. The cmdFromGround signal feeds the ModeManager with the command received from the ground station. The satState signal groups the set of signals which show the state of the satellite to the ModeManager.
- **Outputs—**The respToGround signal is used to convey responses to the ground station starting from a received command, i.e., cmdFromGround. The actOnSat signal defines the generic set of signals used to steer the components of the satellite in response to the transitions of the cascading FSM implemented in the ModeManager.

This section describes the cascading FSM implemented in the ModeManager. The proposed architecture implements a two-level FSM. The outer finite state machine defines four operating modes, i.e., PWR_UP, INIT, HOLD, and NOM. Each operating mode defines a specific overall state of the satellite and the corresponding available actions. For each operating mode, the ModeManager implements a custom finite state machine to execute the actions.

Definition of the operating modes— Figure 10 shows the top view of the FSM as composed of four operating modes, i.e., PWR_UP, INIT, HOLD, and NOM. The PWR_UP operating mode is a temporary wait state used to let the board power up properly. At the end of the brown period, the FSM moves into the INIT state. In the current implementation, the PWR-UP operating mode is a placeholder for future development, and its traversal is instantaneous. The INIT operating mode oversees the initialization procedures for all the peripherals of the satellite. At the end of the INIT, the FSM moves to the HOLD waiting for a command from the ground station before moving the system to the nominal operating mode, i.e., NOM. The HOLD mode represents the debug state of the cascading FSM to provide a communication link to the ground station to inspect the state of the satellite,

solve anomalies, and send specific commands from the ground to the satellite. Notably, the HOLD mode is the destination operating mode at the end of the procedures of INIT or in case of errors during the operations of the satellite. The NOM mode defines the operating mode where the satellite executes the commands from the schedule received from the ground station.

Transitions of the operating modes—At power up, the outer FSM enters the PWR-UP mode. At the end of a predefined time period, i.e., pwrUpTimeElapsed, the outer FSM moves to the INIT mode to perform the initialization of the peripherals of the satellite in terms of communication links, ADCS, and solar panels. The outer FSM moves to the HOLD operating mode at the end of the procedures of INIT. Notably, a command from the ground is required to move the outer FSM from the HOLD mode to the NOM one (see cmdFromGround signal in Figure 10). Once in the NOM mode, the ModeManager performs the operations according to the schedule received from the ground and returns to the HOLD mode due to errors or exceptional behaviors.

Interlocked transition mechanism and FSM structure—The cascading FSM implements a finite state machine for each operating mode. To ensure consistency between the currently active operating mode and the executed finite state machine, the cascading FSM implements an interlocked transition mechanism. Each FSM has a unique entry point state, and its name is obtained as the concatenation of the name of the corresponding operating mode and the string _INIT. For example, INIT_INIT is the entry state INIT's FSM. For each FSM, the entry state is a dummy state that only checks for the transition of the cascading FSM to the correct operating mode. Once the operating mode has been correctly updated the FSM moves to the next state. Notably, the transition of the state happens first and the operating mode one follows. The interlocked transition mechanism is meant to strengthen the relation between each operating mode and the corresponding set of executed actions. The rest of this part details the operations within each macro state.

INIT operating mode—The INIT operating mode oversees the procedures to boot the communication system, starts the ADCS and deploys the solar panels.

Figure 11 shows the state diagram of the finite state machine that implements the INIT operating mode. At the end of the interlocked transition mechanism, the FSM enters the COMM_BOOT state. A timer is set, and the logic of the state tries to boot the communication system. The FSM moves to the COMM_DEPL state when the timer expires, regardless of the outcome of the COMM_BOOT procedures. Such behaviour is meant to allow multiple attempts to power up the COMM module, still capping the time spent in the state. Notably, the solar panel deployment, i.e., DSA_DEPL state, must be entered within a fixed time period to start recharging the batteries, thus avoiding aborting the mission. The COMM_DEPL state oversees the procedures to deploy the communication if the boot of the COMM module was successful, acting as a bypass state otherwise. The logic of the COMM_INIT state is meant to activate a task to periodically send a beacon before moving the FSM to the ADCS_BOOT state. The ADCS_BOOT and ADCS_DTMB states oversee the activation of the ADCS module. After the ADCS power-up procedure in the ADCS_BOOT state, the FSM moves to the ADCS_DTMB state that oversees the actual module activation. A timer is set at the beginning of the ADCS_BOOT state. The timer sets the maximum time to perform the power-up and the ADCS_DTMB of the ADCS before deploying the solar panels, i.e., DSA_DEPL. In case of errors in the ADCS_DTMB state, the FSM tries to reboot the ADCS while the timer keeps decreasing. When the timer expires, the FSM moves to the DSA_DEPL state regardless of the state of the ADCS. It is important to note that the procedures within the INIT operating mode are executed in strict order, even in the case of errors. Any raised error is accounted by setting the corresponding flags in the status register of the ModeManager. At the end of the INIT procedures, the FSM enters

the HOLD operating mode that allows for checking the status of the components of the satellite. Notably, the specification requires that the FSM of the INIT operating mode is executed once across the entire life of the satellite, i.e., across reboots and/or power-on cycles. To implement such behaviour, the computing platform must offer a permanent storage component that allows storing the information INIT_DONE at the end of the first execution of the INIT FSM. Such information is then read at the beginning of the FSM of INIT, allowing the ModeManager to conditionally skip the execution of the FSM. The actual software implementation has been changed in two states of the FSM of INIT. First, additional instructions have been added at the end of the logic implementing the interlock mechanism in the INIT_INIT state. The additional logic implements a reading function, i.e., readPeripheral(), to get persistent information (INIT_DONE) before moving to the next state of the FSM of INIT. If the information is set, i.e., the INIT FSM has been executed in the past, the ModeManager moves directly to HOLD_INIT also changing the operating mode. If the information is clear the ModeManager continues the execution of the INIT FSM. Second, additional instructions have been added to the final state of the FSM of INIT, i.e., DSA_DEPL, to set the INIT_DONE information in the permanent storage component.

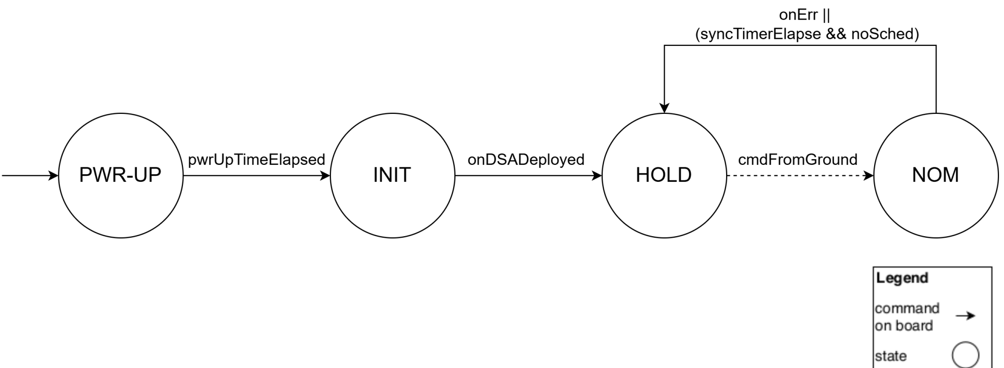

Figure 10. State diagram of the operating modes of the ModeManager.

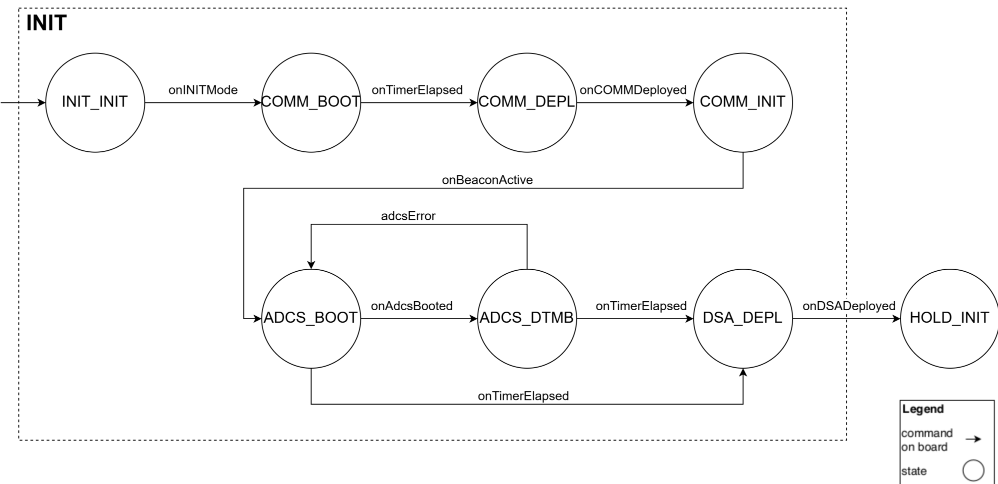

Figure 11. State diagram of the finite state machine of the INIT operating mode.

The HOLD operating mode—At the end of the procedures within the INIT operating mode, the cascading FSM enters the HOLD operating mode. Notably, the system can also enter into the HOLD operating mode in case of errors during the execution of the procedures in the NOM operating mode. As for any operating mode transition, the cascading FSM enters the INIT state, then it waits for the transition of the operating mode to complete the

interlock transition mechanism. At the end of the interlock transition mechanism, the FSM moves from the HOLD_INIT to SETUP, thus starting the execution of the procedures related to the HOLD operating mode as depicted in Figure 12.

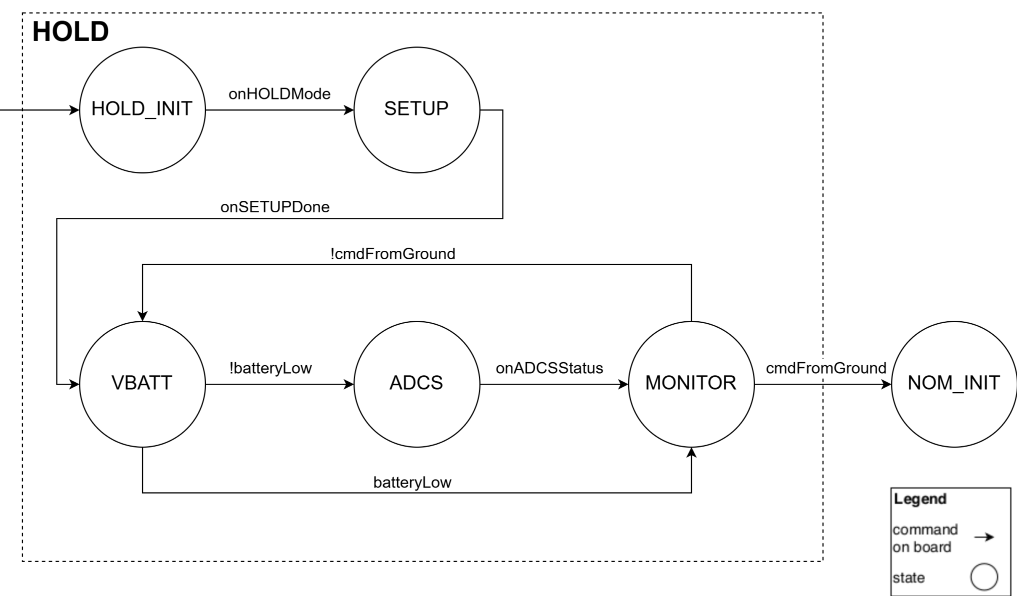

Figure 12. State diagram of the finite state machine of the HOLD operating mode.

SETUP oversees the procedures that must be executed once for each transition to the HOLD operating mode. At the end of SETUP, the FSM loops across three states, i.e., VBATT, ADCS, MONITOR, until a specific command from the ground moves the FSM to the NOM operating mode. The VBATT state checks the state of the battery. If the battery is low the FSM enters the MONITOR state; otherwise, the FSM enters the ADCS state. The ADCS state checks the status of the ADCS before moving to the MONITOR state. The MONITOR state performs two actions. First, it checks the status of all the other components. Second, it decodes any command from the ground to move the FSM to the NOM operating mode. The importance of constantly checking the status of the batteries and the ADCS motivates the implementation of two states, i.e., VBATT and ADCS, respectively, while all the monitoring of the remaining components is performed in the MONITOR state. Notably, the FSM keeps looping across the VBATT, ADCS, and MONITOR states until a command from the ground forces the transition to NOM.

The NOM operating mode—The NOM state is at the core of the cascading FSM as it steers the activities of the satellite in nominal operating conditions. In particular, the NOM state performs the following three macro activities:

- Communicates with the ground;
- Orchestrates the execution of the commands in the schedule;
- Continuously monitors the state of the satellite.

Figure 13 details the state diagram of the FSM of the NOM mode of operation. At the end of the interlocked transition procedure between the NOM_INIT state and the NOM operating mode, the FSM moves from NOM_INIT to LISTEN. The LISTEN state is meant to communicate with the ground to receive two types of commands: requests and new schedules. A request is any command that requires the satellite to provide data to the ground, while the schedule represents a set of operations, ordered in time, that the satellite must perform at the end of the listen period. Notably, the listen period happens periodically, and it defines the time window allowing any communication with the ground. From the operative standpoint, a timer is configured when the FSM enters the LISTEN state.

During the listening period, such a timer can be reconfigured by employing a proper request from the ground. When the timer expires, the FSM moves to PROCESS if a proper schedule has been set, moving to HOLD_INIT otherwise. A proper schedule can be either a new schedule received in the current listening period or a previously unfinished schedule. Notably, a new schedule received from the ground overwrites the partially completed schedule, if any. In the LISTEN state, the timer works as a checker to prevent the FSM from remaining in the LISTEN state after the end of the listen period. Moreover, the FSM moves to HOLD_INIT in case any error is raised during the operations within the LISTEN state. The PROCESS state oversees the processing of the schedule whose semantics are discussed later. The commands in a schedule can be either actions or delays. Action requires an actuation on the satellite that is supposed to happen in zero time, i.e., the time consumed to act should be negligible compared to the dynamics of the schedule. A delay requires to setup a timer. Once a delay command is executed, the timer is set, and the FSM enters the MONITOR state (setTimer). Notably, the FSM enters the HOLD_INIT state in case an error is raised during the operations performed in the PROCESS state. The MONITOR state implements a control loop that continuously checks the status of each component in the satellite. In case of an error or a faulty reported status from a component, the FSM moves to the HOLD_INIT state. When the timer expires, i.e., the one set in the PROCESS state, the FSM moves either to LISTEN or back to PROCESS depending on the command that forces the setup of the expired timer. In particular, the FSM moves to LISTEN if the timer has been set as part of the execution of a sync with ground command (syncTimerElapsed). Otherwise, the FSM moves back to PROCESS since the timer has been set as part of the execution of a schedule operation that imposes a delay before continuing the execution (delayTimerElapsed).

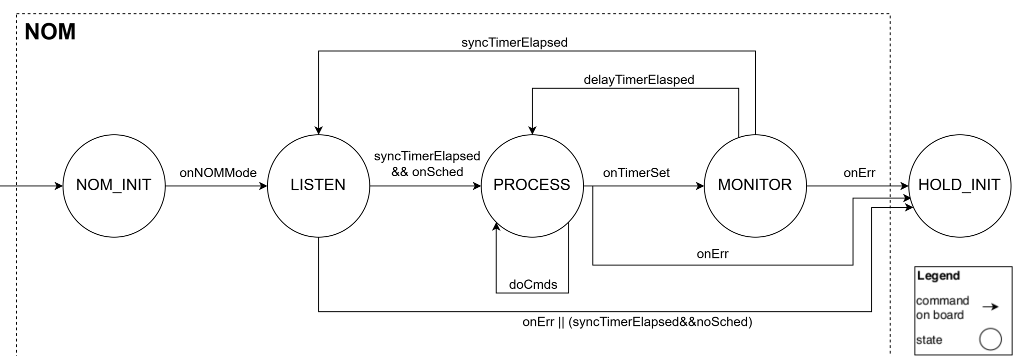

Figure 13. State diagram of the finite state machine of the NOM operating mode.

The `ModeManager` Component

Three components of both the primary and failover deployments compose the Finite State Machine implementation: the `ModeManager` component, containing the custom-made `FSMTransition` and `Procedures`. The `ModeManager` component contains the main logic to drive the execution of the FSM, which can be thought of as 2 layers: *modes* and *states*.

Modes. They are the upper layer: 4 macro-states that mirror the 4 main mission phases (PWR-UP, INIT, HOLD, NOM) (see Figure 10). Transitions between them can be either commanded from ground or result from on-board conditions. The INIT phase shall only be executed once, so after its end `startupMode` parameter is set to HOLD. Furthermore, HOLD mode follows INIT regardless of its outcome, given that the required operations until the first ground contact is established are the same. Nominal mode is entered only if commanded from the ground. The actual mode is saved as a parameter called `Mode`. A component action called `recoverToHOLD` is used whenever HOLD is commanded exter-

nally (ground or FR-FI routines); otherwise, the logic of the component takes care of it. `recoverToNOM` is dedicated to commanding the NOM phase from ground.

States. They are the lower layer: the actual logic as described with the SDL language in Section 3.1. They are implemented only for the INIT and HOLD modes, since PWR-UP is only used to differentiate what mode comes next between INIT (only for the first power-up) and HOLD, so it does not require a complex logic; NOM mode does not have its own logic since during the NOM phase commands will be scheduled from ground. The code implementation for a state is divided into two functions or handlers, an exit and an enter one. The actual—the one in execution—and the next states are saved as parameters called `State` and `NextState`, respectively.

A periodic task of ModeManager, called `update` run, with a chosen period corresponds to a step of the FSM and is composed of two parts: the execution of exit handler of the actual state and the execution of the enter handler of the next state. The latter will become the actual state for the next run of `update`. `FSMTransition` takes care of changing the `NextState` parameter: at the end of the execution of a state the corresponding action of the script is called in `ModeManager` and executed in `FSMTransition`. Instead each action of the `Procedures` component script is a software procedure or an auxiliary action that implements a timer (Timers are implemented as a single `ModeManager Procedures` command *delay* and the time to wait as command argument.). In the enter handler the relevant software procedure of the `Procedures` component is called. In the exit handler, the logic prescribe different outcomes, depending on the conditions set by the counters. If a specific condition is met, there is no state transition but just a call to the `Procedures` component. Instead for the other branches of the logic, `executeStateTransition`, which group together a call to the `FSMTransition` component and another private function needed for the transition, is called and the FSM proceeds as described above.

The schedule. The schedule is a particular message sent from the ground to the satellite during the listen period, and it represents the timed sequence of commands that the satellite must execute at the end of the listen period. The commands in a schedule can be either actions or delays. Action requires an actuation on the satellite that is supposed to happen in zero time; i.e., the time consumed to act should be negligible compared to the dynamic of the schedule. A delay requires setting up a timer. As discussed later, the PROCESS state within the NOM FSM oversees the execution of the commands. The rest of this part discusses the grammar of the schedule and its implementation in ModeManager.

The grammar of the schedule. A schedule starts from the first nonterminal, MSG, which consists of one or more OP separated by the non-terminal SEP (“#” in this case). The OP token represents a command. The separator SEP is a symbol that separates two consecutive OPs. In particular, the separator is removed for fixed-length commands. The proposed grammar supports actions (OP_CODES) and time synchronization with ground (SYNC) commands. Each action requires an attribute whose meaning depends on the actual operating code. Notably, an action can request to set up a time delay to pause the execution of the subsequent commands in the schedule. Each time synchronization with ground command requires a number that encodes the time delay before the actual synchronization.

An example of a schedule generated by this grammar is the following:

```
ine op3 0x01 0x00 0x12 op0 0x00 0x00 NULL timer_op_code 0x01 0x00 0x20 op0 0x01
0x00 0x02 op2 0x01 0x00 0x0A sync_op_code 0x01 0x00 0xFF
ine
```

White spaces are added solely to improve readability; the actual schedule is a sequence of fixedlength commands and attributes without white spaces.

$$\begin{cases} \text{MSG} & \rightarrow (\text{ OP SEP })^{+}\text{OP} \mid \text{OP} \\ \text{OP} & \rightarrow \text{OP_CODE OP_ATTR} \mid \text{SYNC NUMBER} \\ \text{OP_CODE} & \rightarrow \text{op0} \mid \text{op1} \mid \text{op2} \mid \text{op3} \mid \text{op4} \mid \text{op5} \mid \text{op6} \\ \text{OP_ATTR} & \rightarrow \text{STRING} \mid \text{NUMBER} \\ \text{SYNC} & \rightarrow \text{timer_op_code} \mid \text{sync_op_code} \\ \text{STRING} & \rightarrow (a \ldots z \mid \text{A} \ldots \text{Z} \mid \text{NUMBER })^{*} \\ \text{NUMBER} & \rightarrow (0 \ldots 1)^{+} \\ \text{SEP} & \rightarrow \# \end{cases}$$

The example parses as follows: op 3 and op 0 will be executed immediately with attributes 0x12 and NULL, respectively. NULL represents an empty attribute. Then, the timer_op_code 0x20 part of the schedule sets a time delay of 0×20 s. When the timer expires, op0 and op2 will be executed one after the other with attributes 0x02 and 0x0A. The last command in the schedule, i.e., sync_op_code 0xFF, sets a delay of 0xFF seconds before setting up the satellite to receive from the ground station.

The parser of the schedule—The parser implementation features a byte-level `processMsg` function that represents the entry point of the parser. The b_Msg InBuffer contains the schedules as an array of uint8_t, i.e., unsigned integer of 8 bits. The parser decodes each instruction of the schedule by reading the opCode byte. The next two bytes specify the number of bytes to pass as operation argument: the first two bytes of which represent an unsigned integer in little endian order (that is, LSB first) whose value is copied into the variable opNumAttribute. The next $n = opNumAttribute$ bytes are read and stored into the opAttribute variable. Such variable is then processed accordingly inside the called operation specified by the opCode. The current implementation allows the parser to distinguish between three different operations, while additional commands can be easily added by adding else-if statements. The MODEMANAGER_NOM_CMD_EMPTY_BUF operational code defines the end of the schedule. The MODEMANAGER_NOM_CMD_TIMER operational code requires configuring a timer to delay the execution of the next instruction in the schedule. The MODEMANAGER_NOM_SCHEDULE_TIMER requires configuring a timer that forces the transition to the LISTEN state once expired. Then, the specific operations are executed according to the decoded operational code.

The schedule implementation—The ModeManager implements the schedule using a variable-length encoding scheme where the size of the OP_CODE is 1 byte, the number of attributes are 2 bytes and the actual arguments cover the rest of the bytes. To this end, each command within the schedule is encoded using 3 + OpNumAttribute bytes and schedules that includes sync with ground (ground passages) can be sent from the ground to the satellite within a single message. For instance, a schedule can be sent in one message that contains separation bytes by synchronization with ground command. The possibility of encoding long schedules within the same message allows for a single sending from the ground still ensuring the operation of the satellite across multiple passages/listen periods. Notably, the sending of a schedule from the ground overwrites the currently active schedule in the satellite, if any.

3.3. ADCS Mode Manager Implementation

The attitude and determination control subsystem (ADCS) is governed by a low-level mode manager that executes cyclically on the on-board computer. Its responsibilities are threefold:

- Monitor the health and availability of sensors and actuators;
- Trigger safe recovery actions on anomalies;
- Select and initialise the appropriate ADCS control mode as commanded by the main mission FSM, while respecting the current set of available sensors/actuators.

In addition, the mode manager persists its configuration to non-volatile storage and restores it at boot to ensure deterministic behaviour after reboots or power cycles.

3.3.1. Sensor Monitoring and Safe Recovery

At each cycle, the mode manager logic pools a set of hardware status flags for Sun sensors, IMU and gyroscopes, magnetometers, GNSS receiver, magnetorquers, reaction wheels, thermocouples, and power interfaces [19]. For each device class it maintains error and recovery counters and, upon detecting a fault, executes a dedicated safe procedure. Faults are detected from missing communication acknowledgments, error flag rises, anomalies in measurements as detected from ADCS algorithms, and power channel statuses [20]. These procedures are minimally invasive and tailored to the device:

- Power-cycle or reinitialize the affected device (e.g., toggle power channel or power-on switches, then retry identification/measurement).
- Fall back to redundant sensing paths (e.g., switch from IMU angular rates to standalone gyroscopes, or from fine Sun sensors to the coarse Sun sensors) when primary data remain unavailable after retries.
- Derive availability flags from the set of individual device statuses, and expose them to the control logic.

If the manager determines that essential measurements are simultaneously unavailable (e.g., no Sun and magnetic field information) or both actuation branches are lost (e.g., no magnetorquers and insufficient active reaction wheels), it raises a request for hardware safe mode to the main FSM and marks the ADCS status as in *fatal-error*. Otherwise, it keeps attempting local recovery and continues operating in a degraded but controlled state.

3.3.2. Mode Selection and Initialisation Under Constraints

Mode transitions are driven by a commanded *macro-mode* coming from the main FSM (e.g., detumbling, Sun pointing, inertial pointing, communication mode, science mode, etc.). On reception of a new command, the manager:

1. Recomputes availability of sensing/actuation resources from the latest status.
2. Applies a rule-based mapping from the commanded macro-mode and the availability flags to a concrete control initialisation sequence. For instance:
    - If reaction wheels are available, Sun pointing is initialised with momentum management control algorithms; otherwise, the manager enables magnetorquers and selects a magnetic control configuration.
    - If Sun sensors are not available, a commanded Sun-pointing mode is automatically degraded to a safe solar-acquisition variant that relies on solar arrays power data [21].
    - If GPS or gyroscope measurements are missing, full-Sun modes are degraded to direct Sun-pointing.
3. Executes a hardware bring-up sequence consistent with the chosen control law (e.g., power channel enables, and power-on lines for individual units, sensor configuration, etc.), with timing guards between steps.

4. Programmes the ADCS algorithms parameters (e.g., mode number, control block, feature toggles such as continuous desaturation enable, albedo corrections, calibration flags) and confirms that the algorithm is running.

This approach ensures that the commanded objective is honoured when possible and gracefully degraded when prerequisites are not met, while keeping the main FSM agnostic of low-level hardware contingencies.

3.3.3. Configuration Persistence and Restoration

The manager maintains a compact configuration structure containing: (i) supervisory flags (e.g., watchdog enable, time-keeping synchronisation), (ii) last known ADCS status, (iii) per-device status masks and counts, (iv) ADCS algorithms settings (e.g., requested mode, feature toggles, active wheel set, desaturation indices, etc.), and (v) guidance target references (e.g., angular-rate, quaternion, and ground-station targets). The structure is serialised to non-volatile storage as a set of byte and word channels.

On power-on or reboot, the manager attempts to read the stored configuration; if any field is invalid or uninitialized, defaults are applied and a log event is raised. Fatal ADCS status at boot triggers writing defaults and re-reading to ensure a clean baseline. After restoration, the configuration is mirrored back into the live variables and ADCS bus, so that algorithm parameters, active device masks, and last valid macro-mode are consistent with the recovered state. To avoid resuming from unstable transients, if the last mode was a manoeuvre state lacking a persistent attitude profile (e.g., slewing), the manager forces a safe acquisition mode (e.g., solar acquisition) as the starting point.

Following a successful start and a few healthy cycles, the manager marks the running flight image as *stable* via the boot control interface. This prevents unnecessary rollbacks and provides traceability for subsequent configuration commits.

By combining device-specific safe procedures, availability-aware mode mapping, and robust configuration persistence, the low-level ADCS mode manager provides deterministic recovery from common on-orbit faults while honouring high-level mission commands. It confines complexity to the ADCS boundary, reduces the number of special cases exposed to operations, and shortens the ground validation loop for corrective updates.

4. Continuous Development and Testing

The digital twin facility is not limited to the early design and verification phases but extends its value throughout the entire spacecraft lifecycle. Its modularity and reconfigurability enable continuous development and testing of both hardware–software interfaces and embedded flight software, offering a powerful environment for iterative validation and regression testing, as in Figure 14.

One of the main advantages lies in the capability to rapidly accommodate modifications in the spacecraft hardware configuration. For instance, when a change occurs in a peripheral component—such as the replacement of a sensor model or the introduction of a new actuator driver—the corresponding software interface implemented on the microcontroller can be updated in C code to reflect the new communication protocol or message structure. Without altering the overall setup, the updated interface can be seamlessly integrated into the existing digital twin facility, ensuring that the On-Board Computer (OBC) continues to interact with a faithful representation of the flight hardware.

Similarly, the high-fidelity simulator provides the flexibility to evolve along with mission requirements. Environmental or dynamical models can be refined to include higher-order effects, new payload constraints, or alternative actuation strategies. By updating the MATLAB/Simulink models embedded in the simulation kernel, developers can

https://doi.org/10.3390/aerospace13010055

immediately assess the impact of such modifications on OBSW performance in real time, preserving consistency with ECSS-compliant simulation standards.

This flexibility also extends to the Input/Output (I/O) structures towards the emulated hardware. For example, if a communication bus requires a different framing convention, checksum scheme, or timing specification, the digital twin facility allows developers to implement and validate these modifications at the protocol level. The OBC can then be re-tested against the new configuration without requiring physical re-wiring or dedicated test benches, accelerating the verification loop.

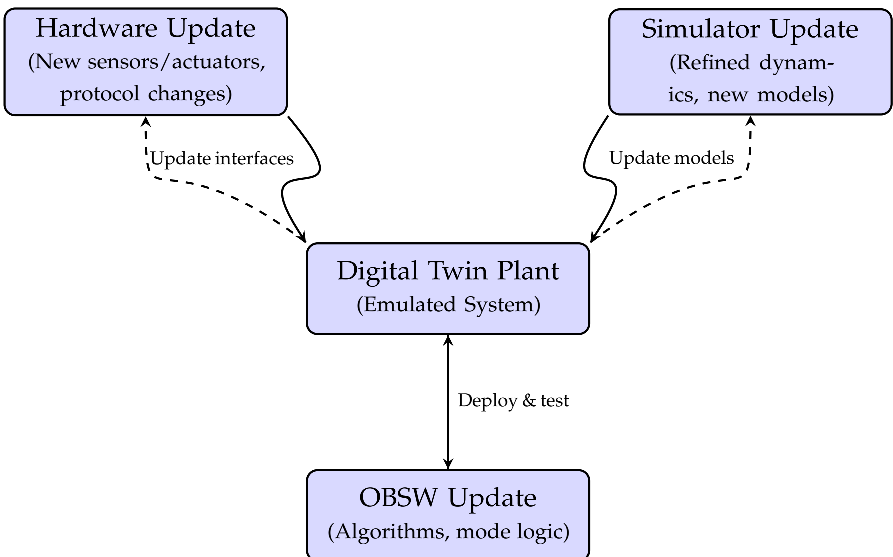

Figure 14. Continuous development and testing loop: updates to hardware, simulator, or OBSW can be integrated and immediately re-verified on the same digital twin plant. Dashed arrows indicate feedback of verification results.

On the software side, any update of the OBSW itself—whether related to Guidance, Navigation, and Control (GNC) algorithms, platform mode management logic, or fault detection and recovery routines—can be re-deployed onto the OBC and validated against the same digital twin representation of the spacecraft system. This guarantees that new software releases are systematically tested under nominal and fault-injected scenarios, with regression checks ensuring that previously verified functionalities remain intact.

Practical Examples

The following non-exhaustive examples highlight how the continuous development and testing philosophy has been applied in practice:

- **Reaction wheel driver update:** If a supplier revises the communication protocol of a reaction wheel unit, the microcontroller code defining the message framing can be quickly updated in the digital twin facility. The OBC can then be re-tested against the new driver interface without requiring access to the physical actuator.
- **Sun sensor replacement:** Should a change occur in the platform design requiring substitution of the sun sensor hardware with a different vendor component, the corresponding sensor model in the high-fidelity simulator can be replaced and calibrated, while the microcontroller interface is updated to handle the new data format. The digital twin thus ensures continuity in the verification process, despite hardware substitution.
- **Navigation function testing:** The same emulated sensor suite (magnetometers, gyros, sun sensors, GPS) can be used to test alternative navigation filters, such as an extended Kalman filter versus a multiplicative quaternion filter. By exploiting the

identical virtualized environment, algorithm performance can be compared under equivalent conditions, guaranteeing reproducibility and providing a safe environment for functional trade-offs.

In practice, the digital twin thus serves as a persistent testbed where hardware and software co-evolve. Updates to microcontroller interface code, simulator fidelity, or OBSW logic can be incrementally introduced and immediately re-verified on the same emulated system. This continuity reduces the risk of late integration issues, enhances traceability across development iterations, and ultimately shortens the software release cycle while improving reliability. In this way, the facility fosters an agile development methodology tailored to the constraints of space missions, where robustness and responsiveness to change are equally critical.

5. The Approach Applied

The modular architecture of the digital twin flat-sat facility, together with the processor-in-the-loop (PIL) configuration of the on-board software, enables a systematic approach for validating the software execution and tuning the on-board parameters during the mission development life-cycle. By exploiting high-fidelity dynamics simulation and protocol-level hardware interfaces, the facility reproduces realistic flight environment while allowing controlled manipulation of execution timing, subsystem interactions, and parameter tuning. This realistic simulation facility is also suitable to support in-orbit operations, providing a safe and flexible framework to test corrective actions and validate updates on ground before their deployment in orbit.

5.1. Sensor Calibration and Software Threshold Tuning

A representative use case of the facility concerns the calibration of current sensors and the adjustment of on-board software thresholds. These operations are essential to ensure the coherence of system measurements with the OBSW logical structure.

In this campaign, a hybrid simulation was configured with current sensors directly inserted in the loop between the real hardware and the digital twin environment. The simulator reproduced realistic power flows across the spacecraft subsystems, while the corresponding electrical currents were generated, measured and logged in real-time through the physical sensors under calibration. The objective was to align the measured and simulated quantities, ensuring consistent slopes and offsets across the entire operating range of the sensors.

The experiment reported in Figure 15 covered the full expected operational field, ranging from negative currents (i.e., power inflow to the subsystem) to positive ones. The measured profiles allowed to derive precise gain corrections and to compensate sensor asymmetries, thus improving the consistency of telemetry data. The calibrated responses were then used to define updated on-board software thresholds separating nominal and non-nominal conditions. In particular, current intensities exceeding approximately ± 100 mA were identified as indicative of potential anomalies or subsystem overcurrents.

Once integrated into the software, these threshold values were validated through repeated runs of the hybrid simulation, confirming that the updated parameters correctly triggered alert conditions without false detections under nominal behavior. The procedure demonstrated the capability of the facility to combine electrical calibration and software verification within a unified testing framework, reducing the need for separate laboratory campaigns while ensuring in-flight consistency between physical measurements and virtual telemetry.

https://doi.org/10.3390/aerospace13010055

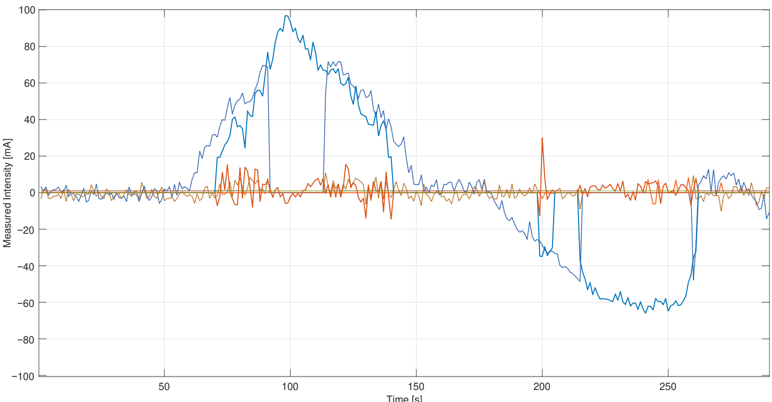

Figure 15. Hybrid simulation with current sensors in the loop. The measured current intensity (negative values correspond to current entering the system) is used to calibrate sensor gains and align measurement slopes across the operational range. The resulting data are employed to tune the on-board software thresholds distinguishing nominal (within ±100 mA) conditions.

5.2. Diverging Tumbling Recovery

A representative case concerns the recovery of a diverging detumbling maneuver after an uncontrolled reboot of the reaction wheels, which occurred without a prior commanded desaturation. This anomaly led to angular rates significantly higher than anticipated and coincided with a degradation in the on-board process execution frequency, slowing down from the nominal 3 Hz to about 1 Hz. Under these conditions, the original control law produced a divergence of the detumbling loop due to a control phase shift, with the angular velocity components undergoing an increasing trend (see Figure 16).

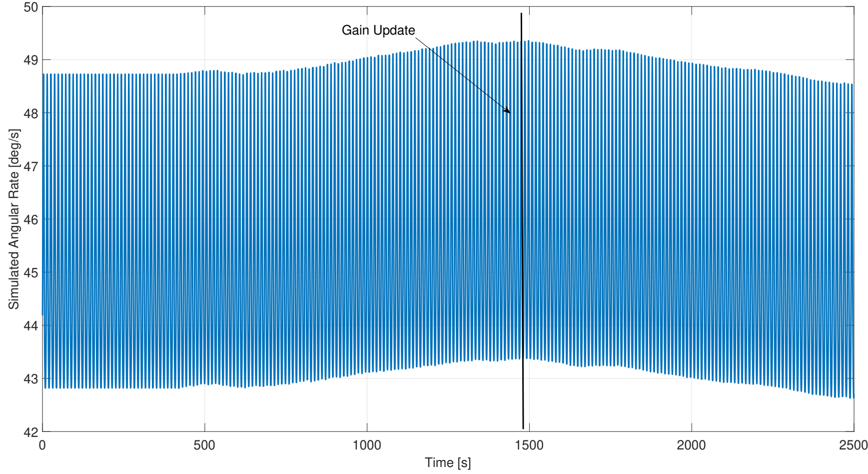

Figure 16. Simulation of the diverging tumbling recovery on digital twin. Processor execution is slowed down to 1 Hz and tumbling conditions are replicated from on-orbit telemetries.

Before commanding recovery action to the space segment, the facility was configured to reproduce the orbital and attitude dynamics together with the actual on-board timing delay, by arbitrarily controlling the execution step and injecting the sudden reaction wheel angular momentum unload. In this way, the simulation successfully replicated the anomaly observed in orbit. Through iterative testing, the control parameters were modified by inverting the sign of the feedback gain and increasing its absolute value. This strategy stabilized the system until the angular rates had decreased to a sufficiently low level, at which point the nominal gain configuration could be safely restored despite the slower execution frequency. Digital twin simulations proved a 3-axis angular rate of about 35 deg/s

as a threshold level to restore nominal gain with the software in contingency running at 1 Hz. The correctness of these values was confirmed by the on-orbit evolution during the commanded recovery action.

Indeed, once validated on the facility, the updated parameters were uploaded in orbit. Telemetry confirmed the effectiveness of the corrective action: the spacecraft angular velocity, which had peaked at more than 200 deg/s after the uncontrolled wheel desaturation event (see Figure 17 and the angular rate step larger than 50 deg/s), was progressively reduced to below 1 deg/s, completing a stable detumbling sequence. The telemetry data in Figure 17 clearly show how the updated gain parameters and the proper timing of telecommand, synchronized with the phase-shift angular rates, produced a stable decay of the angular rates, converging to near-zero values. Once detumbling was achieved, the spacecraft recovered a stable Sun-pointing attitude, ensuring nominal power production and the complete restoration of the battery charge (see Figure 18). This case demonstrates how the facility can reproduce anomalous conditions, identify corrective solutions, and validate software parameter updates prior to their in-orbit execution, significantly reducing operational risks.

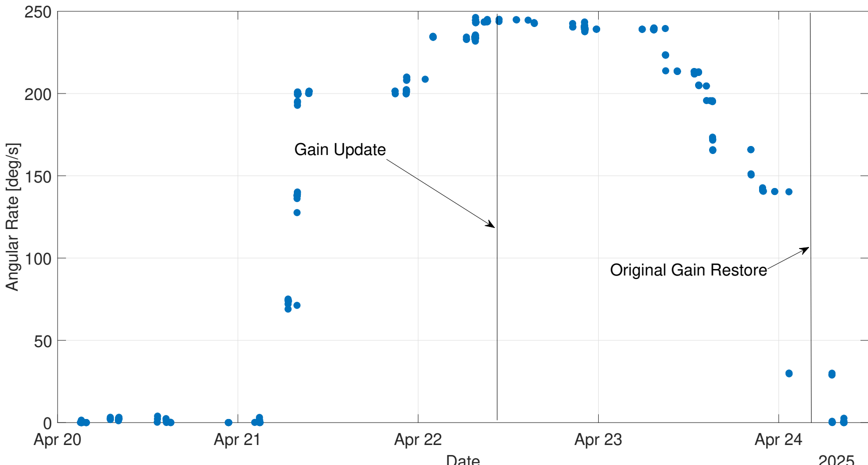

Figure 17. Telemetry data of spacecraft angular rate magnitude across diverging tumbling anomaly recovery.

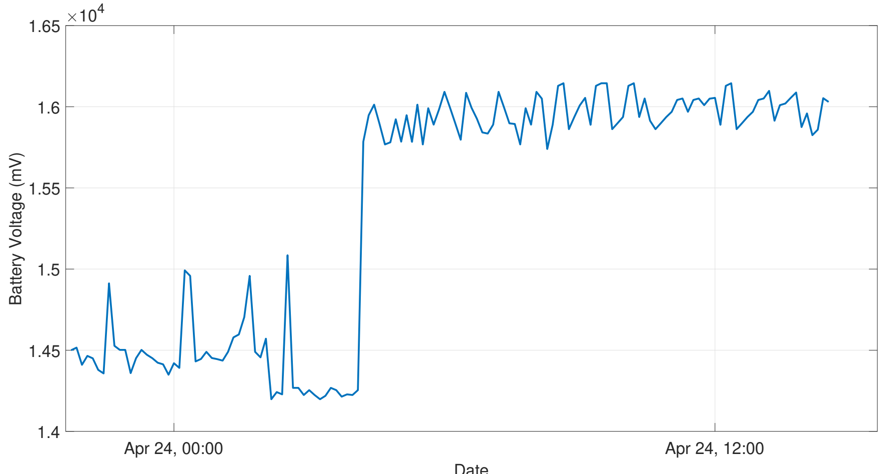

Figure 18. Telemetry data of spacecraft battery voltage after full detumbling has been achieved.

5.3. Power Management at Low Temperatures

Another application of the facility concerned the verification of the spacecraft power management under low temperature conditions, while thermal-vacuum (TVAC) testing

can be performed on ground, these tests are usually focused to assess thermal stability and cycling of the integrated spacecraft. However, the dynamic interaction between orbital power generation, consumption, and thermal boundaries could only be fully reproduced within the digital twin environment.

In this scenario, the facility can be operated in a mixed configuration, where digital models of the orbital dynamics and solar array generation were interfaced with real hardware components sensitive to temperature, most notably the batteries. This setup allows the simulation of worst-case cold conditions, in which battery performance is degraded, while maintaining high-fidelity monitoring of current flows in the flat-sat subsystems and power availability.

The experiments focused on monitoring the balance between peak power consumption and the capability of the simulated solar arrays to sustain the spacecraft operational lifecycle when the batteries were at their lowest allowable temperature. Particular attention was devoted to software-controlled switching of power channels and to the feasibility of handling in-rush currents without violating the safe operating limits of the batteries.

Results (see Figure 19) show that the facility successfully replicated the critical thermal environment and enabled detailed verification of the end-to-end power management logic. The monitored telemetry confirmed that, despite the reduced efficiency of the cold batteries, the system was able to sustain all nominal operations, with the software ensuring correct sequencing of power channel activation and limiting in-rush currents within design margins. This case highlights the capability of the facility to extend power subsystem verification beyond classical thermal tests, providing dynamic validation of software and hardware interactions under realistic environmental constraints.

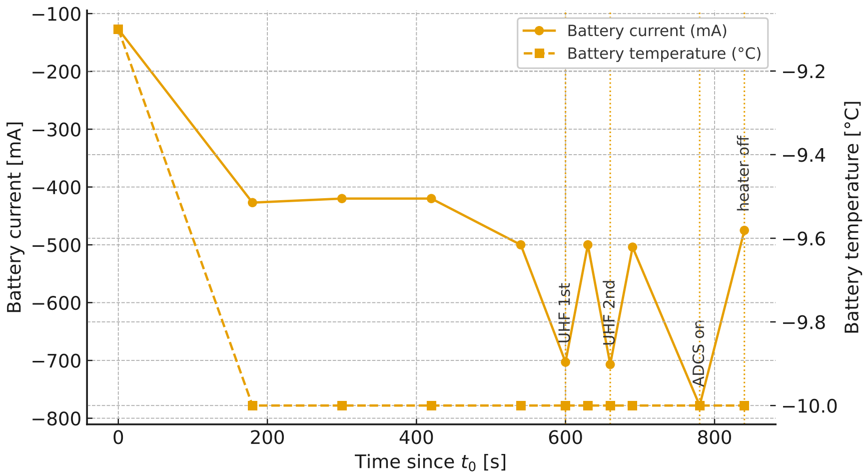

Figure 19. Digital twin simulation of spacecraft power management under low temperature conditions. The digital twin combined orbital solar array generation models with real battery hardware, allowing assessment of current peaks and power channel switching feasibility.

5.4. Quantitative Comparison with Traditional HIL Approaches

While a full Hardware-in-the-Loop campaign remains indispensable for final qualification and acceptance testing, the proposed digital twin framework significantly reduces its required scope by shifting a substantial portion of verification and validation activities to earlier development phases.

From a temporal perspective, the reconfiguration of a test scenario within the digital twin—including updates of the system dynamics, interface modifications, and fault-injection setup—typically requires only hours, as it is performed through software changes in the real-time simulator and in the microcontroller-based interface modules. In contrast, equivalent modifications in a traditional HIL environment generally require days or weeks,

due to hardware availability constraints, physical re-wiring, and the need for repeated re-qualification activities.

In terms of fault coverage, the digital twin enables systematic and repeatable injection of protocol-level faults, timing violations, sensor degradations, actuator anomalies, and software logic errors. Many of these scenarios are impractical, costly, or potentially unsafe to reproduce on physical hardware. As a result, the framework allows a much broader exploration of contingency and failure cases during early development, while preserving HIL testing for high-risk, hardware-critical, or certification-driven scenarios.

Finally, although the digital twin does not aim to replace HIL testing, it substantially reduces the number of HIL test iterations required. This leads to lower overall verification costs and faster feedback cycles, while increasing software maturity prior to hardware-centric validation stages. The combined effect is a shorter time-to-market, a general reduction of mission development costs, and a mitigation of risks associated with late-stage design changes. The additional costs introduced by the digital twin implementation itself are negligible when compared to full HIL campaigns, as the framework is largely software-based, modular, and reusable across different missions and platforms.

6. Conclusions

This work has presented a digital twin-based framework for spacecraft on-board software (OBSW) development, validation, and operations support, which can be applied to modern spacecraft missions. The approach combines a modular flat-sat facility with high-fidelity real-time simulation and microcontroller-based interface emulation, enabling the On-Board Computer and other system components to interact with virtualized subsystems through flight-representative electrical and communication interfaces.

The main findings highlight how this methodology accelerates the software life cycle by allowing early debugging, processor-in-the-loop testing, and systematic verification of critical functions such as finite state machines, fault detection and recovery routines, and Guidance Navigation and Control (GNC) algorithms. By reducing reliance on complex and costly hardware-in-the-loop facilities, the digital twin makes possible more frequent and reproducible test campaigns, including failure injection and recovery analysis, within a safe and controlled environment. From a quantitative perspective, the proposed framework shortens test reconfiguration cycles from days or weeks to hours, expands fault-injection coverage, and reduces the number of required HIL iterations. This translates into faster feedback loops, lower verification costs, and reduced exposure to late-stage design changes, while preserving HIL testing for final qualification and certification purposes.

A key novelty of the presented framework lies in its continuity across the entire mission timeline: the same facility that supports incremental development and integration on the ground can be employed after launch for software updates, anomaly replication, and procedure validation. This establishes the digital twin not only as a development tool but also as an operational asset, enabling in-orbit troubleshooting and performance tuning.

Looking ahead, the integration of this virtualized digital twin platform paves the way toward a new generation of verification and validation practices. Unlike traditional setups, only a fully virtualized environment allows the seamless coupling of real flight processors with hardware-in-the-loop simulations of spacecraft dynamics, anomalies, faults, and contingency scenarios. This capability is essential for testing advanced fault detection algorithms and validating autonomous system reactions under realistic and stressing conditions. As spacecraft become increasingly intelligent and autonomous, such facilities will be instrumental in ensuring resilience and mission continuity even in the presence of multiple anomalies.

https://doi.org/10.3390/aerospace13010055

Overall, the proposed approach offers a scalable and cost-effective approach to spacecraft software engineering, fostering robust design, agile development, and resilient operations. Its contribution extends beyond a single space program, offering a generalizable paradigm for small satellite missions where reliability, flexibility, and efficient verification are decisive for mission success.

Author Contributions: Conceptualization, A.C. and S.S.; methodology, A.C., A.B. and S.S.; software, A.C., A.B. and S.S.; validation, A.C., A.B. and S.S.; formal analysis, A.B. and A.C.; investigation, A.C., A.B. and S.S.; data curation, A.B. and A.C.; writing, A.C., A.B. and S.S.; visualization, A.B. and A.C.; supervision, A.C. and S.S.; revision, A.B., A.C., M.L. and S.S.; funding acquisition, M.L. All authors have read and agreed to the published version of the manuscript.

Funding: Funding: The methodologies developed within this research were internally funded and did not receive any specific external financial support.

Data Availability Statement: Data are not publicly available, but can be provided upon request to the corresponding authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. American Institute of Aeronautics and Astronautics (AIAA) Digital Engineering Integration Committee. *Digital Twin: Definition & Value*; Technical Report; Position Paper; AIAA/Aerospace Industries Association (AIA): Arlington, VA, USA , 2020.
2. AIAA Digital Engineering Integration Committee. *Digital Twin: Reference Model, Realizations & Recommendations*; Technical Report; Implementation Paper; Aerospace Industries Association (AIA)/AIAA/NAFEMS: Arlington, VA, USA, 2023.
3. European Space Agency. *ECSS-E-ST-40C Rev.1: Space Engineering—Software*; Technical Report; ESA Requirements and Standards Division: Noordwijk, The Netherlands, 2021.
4. NASA. *NASA Systems Engineering Handbook (SP-20225009603, Rev 3)*; Technical Report; NASA: Washington, DC, USA, 2022.
5. He, C.; Zhang, Y.; Ke, J.; Yao, M.; Chen, C. Digital Twin Technology-Based Networking Solution in Low Earth Orbit Satellite Constellations. *Electronics* **2024**, *13*, 1260. [CrossRef]
6. Chiti, F.; Pecorella, T.; Picchi, R.; Pierucci, L. Towards Digital-Twin Assisted Software-Defined Quantum Satellite Networks. *Sensors* **2025**, *25*, 889. [CrossRef] [PubMed]
7. Pinello, L.; Giglio, M.; Cadini, C.; De Luca, G.F. Development of a space exploration rover digital twin for damage detection. In Proceedings of the PHM Society Asia-Pacific Conference Proceedings, Tokyo, Japan, 11–14 September 2023. [CrossRef]
8. Hou, Z.; Li, Q.; Foo, E.; Dong, J.; De Souza, P.A., Jr. A Digital Twin Runtime Verification Framework for Protecting Satellite Systems from Cyber Attacks. In Proceedings of the 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS), Hiroshima, Japan, 26–30 March 2022.
9. Pierce, G.J.; Heeren, J.D.; Hill, T.R. ORION SysML model, digital twin, and lessons learned for Artemis I. In Proceedings of the 33rd Annual INCOSE International Symposium, Honolulu, HI, USA, 15–20 July 2023.
10. Corpino, S.; Stesina, F. Verification of a CubeSat via Hardware-in-the-Loop Simulation. *IEEE Trans. Aerosp. Electron. Syst.* **2014**, *50*, 2807–2818. [CrossRef]
11. Kiesbye, J.; Messmann, D.; Preisinger, M.; Reina, G.; Nagy, D.; Schummer, F.; Mostad, M.; Kale, T.; Langer, M. Hardware-In-The-Loop and Software-In-The-Loop Testing of the MOVE-II CubeSat. *Aerospace* **2019**, *6*, 130. [CrossRef]
12. Nicolai, A.; Mora, C.M.; Bretfeld, R.; Stroschke, A.; Delayat, V.; Jahn, H.; Raschke, C.; Scheiding, S.; Vidal, P.; Terzibaschian, T. New Developments in Attitude Control Hardware-in-the-Loop Testing. In Proceedings of the ESA GNC-ICATT, Sopot, Poland, 12–16 June 2023 .
13. Di Capua, A.; Candia, S.; Di Cocco, T.; Anania, M. A Software-based Environment for Development & Validation of Spacecraft On-board Software. In Proceedings of the ESA/Industry Conference Proceedings (Indico), Lausanne, Switzerland, 13–15 October 2010.
14. El wafi, I.; Haloua, M.; Guennoun, Z.; Moudden, Z. A framework for developing an attitude determination and control system simulator for Cubesats: Processor-in-loop testing approach. *Results Eng.* **2024**, *22*, 102201. [CrossRef]
15. Colagrossi, A.; Silvestrini, S.; Lavagna, M. Flat-Sat Facility for Processor-in-the-Loop Verification and Testing of Nanosatellite ADCS/GNC. In Proceedings of the ESA GNC-ICATT 2023, Sopot, Poland, 12–16 June 2023.
16. Rizza, A.; Piccolo, F.; Pugliatti, M.; Panicucci, P.; Topputo, F. Hardware-in-the-loop simulation framework for cubesats proximity operations: Application to the milani mission. In Proceedings of the International Astronautical Congress: IAC Proceedings 2022, Paris, France, 18–22 September 2022; pp. 1–15.

17. Glaessgen, E.H.; Stargel, D.S. The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles. In Proceedings of the 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference, Honolulu, HI, USA, 23–26 April 2012.
18. Kul'ba, V.V.; Mikrin, E.A.; Pavlov, B.V.; Somov, S.K. A Comprehensive Software Verification Technology for Onboard Control Systems of Spacecraft. *Autom. Remote Control* **2023**, *84*, 1047–1054. [CrossRef]
19. Colagrossi, A.; Lavagna, M.; Bertacin, R. An Effective Sensor Architecture for Full-Attitude Determination in the HERMES Nano-Satellites. *Sensors* **2023**, *23*, 2393. [CrossRef] [PubMed]
20. Colagrossi, A.; Lavagna, M. Fault Tolerant Attitude and Orbit Determination System for Small Satellite Platforms. *Aerospace* **2022**, *9*, 46. [CrossRef]
21. Colagrossi, A.; Lavagna, M. A Spacecraft Attitude Determination and Control Algorithm for Solar Arrays Pointing Leveraging Sun Angle and Angular Rates Measurements. *Algorithms* **2022**, *15*, 29. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

https://doi.org/10.3390/aerospace13010055