CUBEX: A CubeSat Exemplar for Teaching Software Architecture Principles

Angelos Motsios[1], Timo Kehrer[2] and Christos Tsigkanos[1,2]

[1]Space Software Group, University of Athens, Greece
[2]Software Engineering Group, University of Bern, Switzerland

Abstract. Teaching software architecture effectively requires bridging the gap between theoretical concepts and practical application, particularly in complex domains like software for aerospace systems. This paper presents CUBEX, an exemplar developed for teaching software architecture to master's level university students. The project utilizes the JPL's F´ flight software framework within the context of a simulated CubeSat mission focused on orientation monitoring. Students are tasked with developing key flight software components, specifically an Inertial Measurement Unit driver and a Payload processing component, adhering to specified mission requirements. Through this process, students gain practical experience with core software architecture principles, including component-based design, interface definition, telemetry and event handling, and system integration. We detail the project's technical foundations, pedagogical structure, and its value as a readily available, reusable educational artifact designed to facilitate teaching of contemporary software architecture principles.
Artifact: software.aerospace.uoa.gr/cubex
Video: software.aerospace.uoa.gr/cubex/overview.mp4
Keywords: Flight Software Architectures · Architecture Education

1 Introduction

Software architecture stands as a critical discipline within software engineering, dictating the fundamental structure, interactions, and properties of complex systems. Educating future software architects [10] –particularly at the master's level – necessitates pedagogical approaches that move beyond abstract lectures towards tangible, hands-on experiences [1]. Project-based learning, especially when grounded in realistic domains like aerospace flight software (FSW), offers a powerful mechanism for students to grapple with architectural trade-offs, design patterns, and implementation challenges. However, educators often face a scarcity of well-structured and accessible exemplars that utilize modern frameworks representative of industry practice; the significant effort required to create comprehensive teaching material, including setup guides, tutorials, and realistic requirements, often presents a barrier. Architecture education [7] often struggles to bridge the gap between theoretical concepts and practical application.

A real-world domain highlighting that a practical skillset is instead required is the engineering of flight software for on-board space systems, which involves

a constellation of complex architectural challenges [6]. Those include constraints imposed by limited computational resources, the imperative for high reliability as well as intricate operational demands of particular missions; sophisticated and well thought-out architectural designs are desired [5]. Our overall objective being teaching advanced software engineering concepts, in line with notable efforts in the field [9]. To this end, this paper introduces the CUBEX exemplar, designed to address this need from the perspective of architecture education. It serves as a laboratory exercise project for master's students, focusing on teaching architecture fundamentals within a simulated context. The core task involves building FSW for a simulated CubeSat mission, monitoring the satellite's orientation using real-time accelerometer data.

CUBEX leverages F´ (F Prime [2]), a component-based software framework developed at the Jet Propulsion Laboratory (JPL) and increasingly adopted for small satellite missions and robotics – and famously on the Mars Helicopter. F´ provides a structured environment that inherently promotes key architectural principles like modularity, well-defined interfaces, and separation of concerns, making it an excellent vehicle for teaching software architecture concepts. The exercise context in a space mission adds a layer of realism and relevance for students. The contribution of this paper is the presentation of CUBEX as an educational tool for the software architecture community. It details the project's technical architecture, the specific software architecture concepts inehrent in it, the pedagogical approach embodied in its structured tutorials and requirements, and its availability as an open resource. The subsequent sections cover background on FSW development and the context of the exemplar (Sec. 2), a detailed description of the architecture, requirements (Sec. 3), and how it teaches specific concepts with an overview of the pedagogical structure and learning experience (Sec. 4), and concluding remarks (Sec. 5).

2 Background: FSW Development and Exemplar Context

Understanding the context of CUBEX requires familiarity with both the F´ framework and the specific CubeSat mission scenario employed.

2.1 The JPL F´ Flight Software Framework

F´ is an open-source framework designed for the rapid development and deployment of flight software systems. Originating from NASA/JPL, it rigorously follows a component-based architecture. Its key tenets, in brief, entail:

- **Components:** The fundamental units of design and modularity in F', components are analogous to C++ classes – they encapsulate specific functionalities and interact with each other through well-defined interfaces.
- **Ports:** Typed interfaces used by components for communication, defining the data types and directionality of interactions.
- **Topology:** A model representing the overall system architecture, specifying component instances and the connections between their ports.

- **Commands, Telemetry, Events :** Built-in mechanisms for space-ground interaction, data reporting, significant event logging, and configuration management, respectively are key elements foundational in the space context.
- **Modeling and Autocoding:** F´ utilizes a domain-specific modeling language (FPP – F Prime Prime [3]) to define components, ports, types, and topology. This model is then used by autocoding tools to generate significant portions of target C++ code, including serialization, connection logic, etc.

The suitability of F´ for teaching software architecture stems directly from these features. F´ enforces modular design through components, requires explicit interface definition via ports and the FPP topology, promotes separation of concerns (e.g., application logic versus communication), and provides concrete mechanisms for common FSW patterns like telecommand handling and telemetry. The C++ implementation base provides exposure to a language widely used in embedded systems, while the autocoding aspect introduces students to model-driven development practices. Furthermore, its provenance and use in real missions enhances student engagement.

2.2 CubeSat Simulated Mission Scenario

The project is situated within a specific – albeit simplified – CubeSat mission scenario. This context provides a narrative and concrete goals.

Mission Objective: The primary goal entails monitoring a CubeSat's orientation and movement in space using data from an on-board accelerometer. Software must acquire acceleration sensor data along three axes in real-time and transmit it to the ground station. The FSW is required to continuously acquire accelerometer data, compute average acceleration over the last second for each axis, determine the dominant axis of acceleration and provide real-time feedback to ground facilities on the CubeSat's operational state and orientation.

The target CubeSat scenario serves a crucial pedagogical purpose – it transforms abstract architectural concepts into tangible requirements (outlined later in Sec. 4). For instance, "telemetry" becomes specifically accelerometer data, "events" become dominant axis change notifications, and component interactions are driven by the need to process sensor data and control physical actuators. For the latter, LEDs are used in place of more advanced alternatives used in actual missions – a decision also to enable breadboarding as an initial electronics bootstrapping exercise. The grounding in a relatable mission makes the architectural constructs easier for students to understand and motivates the design choices needed to fulfill the mission objectives, as explicitly captured in requirements.

3 CubeSat Examplar for Teaching Architectural Concepts

CUBEX intends to provide a structured environment and materials for students to learn software architecture principles through practical implementation.

3.1 CUBEX Flight Software Architecture

The central goal for students undertaking the exercise is to develop and integrate specific components of the CubeSat's flight software using the F´ framework. A detailed walkthrough is provided to students, along with drivers and example code that does not address core tenets of the project, to lower the on-boarding overhead of dealing with low-level embedded code such as device drivers.

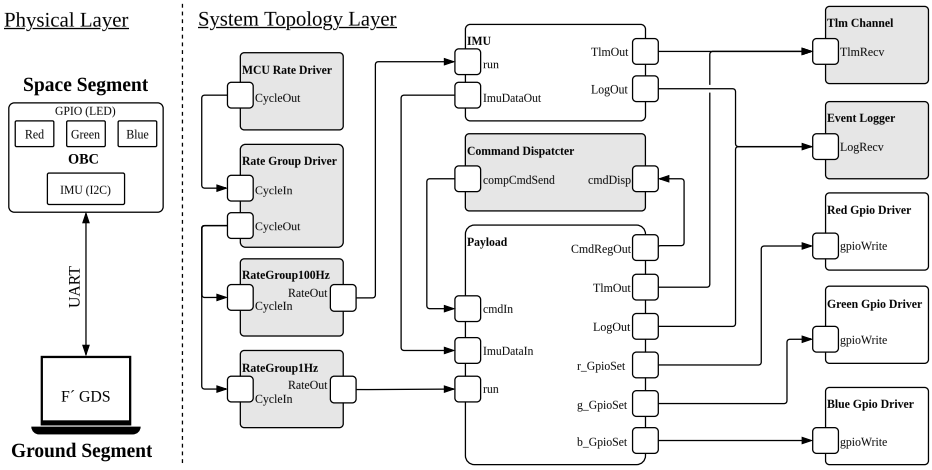

Fig. 1. CUBEX architecture in two layers: (a) Physical Layer demonstrating integration of the IMU sensor with the MCU for data acquisition, with LED outputs for status indication and the Ground Data System. (b) System Topology Layer demonstrating the target architecture. Marked in gray are F´ core framework components.

The overall system architecture consists of multiple interacting F´ components deployed onto a microcontroller platform utilizing a dual-core ARM Cortex-M0+ MCU at 133 MHz, featuring 264 KB of on-chip SRAM, augmented with an MPU6050 accelerometer sensor connected via I2C, and according supporting electronics. MCUs at this class typically serve the capacity of the On-Board Computer (OBC) in a space mission and are responsible for core FSW tasks. Within CUBEX, students primarily focus on understanding the functionality and developing two discrete components:

IMU Component: This component is responsible for interfacing with an Inertial Measurement Unit (IMU) sensor, by providing data regarding the spacecraft's linear acceleration at a specific sampling rate. Students learn to integrate drivers into the F´'s build system to achieve communication with the IMU and interpret raw sensor data. Such data will later be processed and transformed into telemetry and subsequently passed through a defined F´ port interface into the next processing component within the FSW architecture.

Payload Component: Building upon the reception of linear acceleration data from the IMU component, the Payload component is specifically designed to first read and then store incoming data. The data for the X, Y, and Z axes is organized and held within a vector structure accessible through the component

port. A key student task consists to implement a running average algorithm that operates on these stored acceleration values for each axis over a one-second window with a specific sampling rate. The resulting average acceleration values is then formatted and transmitted as telemetry. Following this, the component analyzes these averaged values to identify the dominant axis of acceleration.

Observe that the division of responsibilities between the IMU and Payload components directly illustrates architectural principles like separation of concerns (hardware interaction vs. business logic) and modularity, while the interaction mode between them exclusively via F´ ports highlights interface-based design.

3.2 Software Requirements

A key element of CUBEX as an educational artifact, is the provision of clear and itemized requirements that need to be satisfied by a design – in line with software requirements in the aerospace domain. These requirements bridge the gap between the high-level mission objectives and the specific functionalities to be implemented in software by student teams. Requirements follow a structured naming convention (Module-Component-Number), hinting at traceability practices as illustrated in Table 1[1].

Table 1. (Fragment of) Software requirements of CUBEX [1].

Requirement ID	Description
Comp-IMU-3	Components::Imu shall be able to produce telemetry and events for the IMU's I2C status.
Comp-IMU-4	Components::Imu shall produce telemetry of accelerometer data at 100Hz.
Comp-Payload-2	Components::Payload shall compute the average acceleration for the X, Y, and Z axes over the last 1 second and produce telemetry.
Comp-Payload-3	Components::Payload shall determine the dominant axis of acceleration based on the computed average values and emit an event.

Observe that these detailed requirements serve as essential *pedagogical scaffolding*. Instead of providing vague instructions, they break down the overall task into smaller, verifiable units of work of increasing difficulty. For example, requirement Component-IMU-3 directly tasks the student with implementing I2C communication, while Comp-IMU-4 specifies the required data rate for telemetry. This structure guides the students' design and implementation efforts, focusing their learning on the specific F´ mechanisms needed to satisfy each requirement and reducing ambiguity (also in evaluation by instructors). It provides clear targets against which students can verify their implementation.

3.3 Illustrating Architecture Concepts

CUBEX is designed as a reusable educational artifact designed to facilitate teaching of contemporary software architecture principles, utilizing a real-world FSW framework. In the following, we highlight key such concepts:

[1] The complete requirements can be found in accompanying material.

Component Modeling and Interface Definition: The exemplar emphasizes the use of high level modeling (here, in the FPP DSL) as the starting point; students learn to define custom data types such as the Vector array used for 3-axis data, or ports to carry acceleration data. This act of defining types and ports in FPP forces students to think about the component's external contract – its interface – before implementation. The tutorial structure guides students through creating components, defining these elements, updating the build system (CMake), and running the appropriate utility to generate corresponding C++ code. This process highlights the model-driven nature of F´ development and pedagogically reinforces the practice of defining architecture and interfaces first, promoting interface-based design and separation of concerns.

Commands, Telemetry and Events: The concepts of telemetry and event logs are central to FSW and are explicitly treated. Within the exemplar, students define commands, telemetry channels and event logs in the FPP model, including their types, severity levels and format strings. Such data are visualized using the built-in F' Ground Data System (Fig. 1), demonstrating how architectural choices enable system monitoring and debugging. In F´ commands, telemetry and events are provided and can be seamlessly integrated within a custom component. Command Dispatcher is responsible receiving encoded command packets, decode them and look up the opcode of the command in a table build by component registrations. Event Logger is responsible for storing and filtering events based on their severity level. Telemetry Chanel is responsible for storing telemetry values in a serialized form in a set of telemetry channels in a table.

Hardware Abstraction and Interaction: Development of the IMU component directly addresses hardware abstraction by encapsulating MPU6050 sensor interaction via I2C using an external library. The provided library includes examples that students can examine to understand the expected behavior. This demonstrates how architectural components can isolate hardware dependencies, making the system portable and testable.

Component Logic: Component's runtime behavior is implemented within C++ handlers, making use of features generated from the model files. For instance, for the Comp-Payload-3 requirement, students have to adopt event methods which will confirm the dominant axis of the IMU. Similarly, Comp-IMU-5 and Comp-Payload-1 (found in accompanying material) target checks of communication between component ports. The explicit connection between the architectural model (in FPP) and the C++ implementation reinforces the benefits of model-driven development and code generation, showing how the framework simplifies implementation by providing hooks consistent with the design.

Topology and Integration: After developing individual components, students learn how to assemble them into a complete system using F´'s topology mechanism. In this crucial step, students must carefully consider the connections defined in the topology, particularly in relation to requirements Comp-IMU-4, and Comp-Payload series 5 to 7 (found in supplementary material). These requirements dictate the execution rate of the IMU component and the connection of the Payload component with the GPIO component driver. The IMU execution rate, as defined by Comp-IMU-4's 100Hz execution, is bound to a Rate-

Group component which determines a specific period at which the components connected will execute. A key highlight is the reusability of the RateGroup component itself. Multiple components, not just the IMU, that require the same execution frequency can be connected to the same RateGroup instance. The RateGroup component takes the Cycle rate from the RateGroupDriver component, which acts as a divider of the hardware's tick rate in order to adjust the triggering of different components, further enhancing reusability by allowing a single RateGroupDriver to serve the timing needs of numerous RateGroup instances operating at various frequencies. Integration shows how independently developed components are composed and how system-wide concerns like timing are managed architecturally, forcing students to consider how components interact to achieve overall mission goals.

4 Pedagogical Structure and Experience

The exemplar is designed not just as a technical exercise but with a clear pedagogical structure aimed at maximizing learning outcomes for master's level students, assuming CS background[2].

Learning Objectives. Upon successful completion of the exercise, students are expected to achieve the following learning objectives:

- *Understand fundamental architectural concepts* of the F´ framework, including components, ports, topology, commands, telemetry, and events.
- Gain practical experience in modeling F´ components using FPP and implementing their behavior in C++.
- *Apply core software architecture principles* such as modularity, interface-based design, and separation of concerns within a realistic project context.
- *Develop an understanding of how software components interact* with hardware peripherals (sensors and actuators) within a structured framework.
- *Learn the importance of requirements* and gain experience in implementing software to meet specific, documented requirements.
- *Become familiar with the development workflow* typical in embedded and flight software, involving modeling, code generation, implementation, system integration and testing.

Mapping Exemplar Features to Architectural Concepts. To provide a clear link between hands-on experience and the underlying architectural knowledge being taught, Table 2 maps specific practical features (or activities) within CUBEX to corresponding, broader theoretical architectural principles and practices that these features illustrate.

5 Conclusions and Future Work

The paper introduced CUBEX, an exemplar designed for teaching software architecture principles to master's-level university students. The exemplar uses

[2] CUBEX has been adopted at the MSc-level "Space Software" course at the University of Athens and is planned for ""Dependable Cyber-Physical Systems" within the JMCS MSc program at the University of Bern.

Table 2. Mapping features to taught architectural concepts.

Features of exemplar	Teaching architectural concepts
Project Requirements	Requirements Engineering, Requirements Traceability
Defining ports and types	Interface Definition, Custom Data Types, Architectural Modeling (FPP), Build System
Defining telemetry and events	Observability Architectural Modeling (FPP), Hardware Abstraction
Component logic development	C++ implementation, Hardware Interaction, Framework API Usage
Full System Integration	System Assembly, Component Instantiation, Scheduling, Component Reusability

the JPL's F' [2] framework within a simulated CubeSat mission, where students develop key flight software components, gaining practical experience with core software architecture principles. We plan to integrate more advanced software engineering concepts as elective extensions to the exemplar such as formalization of requirements [4], integrating runtime verification and goal modeling [8], as well as scaffolding for accommodating other payloads such as sensors, motors etc, covering wider architecturally-backed software engineering aspects [6], and reporting on the respective educational experience in depth.

Data Availability Statement. Supplementary material can be accessed at: software.aerospace.uoa.gr/cubex. **Acknowledgements.** Partially supported by HFRI/Greece Project 15706 and SNSF/Switzerland Project 220875.

References

1. ACM/IEEE/AAAI Joint Task Force on Computer Science Curricula: Computer Science Curricula 2023 (January 2024)
2. Bocchino, R., Canham, T., Watney, G., Reder, L., Levison, J.: F prime: An open-source framework for small-scale flight software systems. 31st AIAA Conf. (2017)
3. Bocchino, R.L., Levison, J.W., Starch, M.D.: Fpp: A modeling language for f prime. In: 2022 IEEE Aerospace Conference (AERO), pp. 1–15 (2022)
4. Bögli, R., Rohani, A., Studer, T., Tsigkanos, C., Kehrer, T.: Temporal logics meet real-world software requirements: A reality check. 13th International Conference on Formal Methods in Software Engineering (FormaliSE) (2025)
5. Dvorak, D.: NASA Study on Flight Software Complexity. In: AIAA Infotech@Aerospace. American Institute of Aeronautics and Astronautics (2009)
6. European Cooperation for Space Standardization: ECSS-E-ST-40C: Space Engineering - Software (2009)
7. Kiwelekar, A.W., Wankhede, H.S.: Learning objectives for a course on software architecture. In: Software Architecture (2015)
8. Li, J., Tsigkanos, C., Li, N., Tei, K.: Instrumenting Runtime Goal Monitoring for F' Flight Software. In: 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1300–1309 (2024)
9. Rozier, K.Y.: On teaching applied formal methods in aerospace engineering. In: Formal Methods Teaching. Springer (2019)
10. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software architecture: a comprehensive framework and guide for practitioners. Springer (2011)