Space Flight Software Systems: An approach to understanding their Open Source Framework Paradigm

Technical Report · March 2022

DOI: 10.13140/RG.2.2.12451.94244

2 authors:

Thibault Farges
SODERN
1 PUBLICATION **1** CITATION
SEE PROFILE

Ugo Levi Cescutti
European Institute of Technology
1 PUBLICATION **1** CITATION
SEE PROFILE

Space Flight Software Systems:

An approach to understanding their Open Source Framework Paradigm

Thibault Farges[1], Ugo Levi--Cescutti[2]

[1] SODERN Arianegroup, Software Engineering Department, Space Flight Software Branch
[2] EPITECH, school of Software expertise

Author Note

Thibault Farges,
LinkedIn: https://www.linkedin.com/in/thibault-farges-7a8944ba/
GitHub: https://github.com/ThibFrgsGmz

Ugo Levi--Cescutti,
LinkedIn: https://www.linkedin.com/in/ugo-levi-cescutti
GitHub: https://github.com/ugo94490

First edition of this manuscript completed in March 2022

Any correspondence regarding this article should be addressed to Thibault Farges (email: farges[dot]thibault[at]gmail[dot]com

How to cite this report:
FARGES Thibault, & LEVI--CESCUTTI Ugo. (2022, march). Space Flight Software Systems: An approach to understanding their Open Source Framework Paradigm.

Legal Notice

Abstract

The democratization of access to space necessitates lowering the cost and time to market of software solutions for applications and services. Open source frameworks are one of the best approaches to addressing these challenges. They enable software system standardization and mutualization while serving as a vehicle for innovation and knowledge sharing. The research defines the space flight software framework paradigm to bring stakeholders together in an open approach and discover new opportunities towards an inclusive environment prepared to face the realities of today's economy. Characterizing this paradigm required a refresher on the major guidelines for spaceflight software system architectures and their quality attributes, as well as defining what a framework should strive for. We created an open source framework evaluation system and used it to evaluate three popular frameworks: F Prime (NASA JPL), cFS (NASA GSFC), and TASTE (ESA ESTEC). The results suggested that frameworks should be viewed as tools in which the user should be as independent as possible, or risk being locked into technological dependencies. Each framework was evaluated from an SME perspective, with F Prime and cFS coming out on top, with a preference for the former. Due to the ever-changing software development ecosystem, the paradigm definition will need to be updated to be more evolutionary. System interoperability in terms of message format remains a major concern, with SEDS emerging as the best candidate to address this. Another concern is the long-term sustainability of the frameworks, which requires stakeholders to be more willing to contribute to their expansion. Finally, a suggestion on the importance of an international governance framework to avoid concerns about the digital sovereignty of states.

Keywords: software; framework; space flight; architecture; open source

Table of Contents

Table of Illustrations

1. INTRODUCTION

The democratization of access to space, dubbed "New Space", has enlivened the space sector by allowing new stakeholders, primarily private, to enter the market with disruptive new space applications and services. These new stakeholders always promise more services, imaging capacity, connectivity capacity, or space data applications.

With the growing number of stakeholders in the space community, it is more important than ever to adapt to these upheavals by constantly evolving to develop products faster. These new processes must be built around a few key words: standardize, reuse, centralize, and accelerate. Projects must reduce time to market while maintaining or even reducing costs and preserving or even improving quality.

Stakeholders, whether government agencies, system integrators, or equipment suppliers, are faced with increasingly complex software system requirements for spaceflight, all while working within tight time frames and budgets. Several international working groups, including CCSDS [1], SUMO [2], FACE [3] and SAVOIR [4] advocated for the adoption of a reference architecture to provide a standardized approach to describing software system architectures. These architectures would encourage reuse across multiple missions and provide the business with greater technical responsiveness to inevitable but unpredictable changes [5].

As Uncle Bob [6] would put it, “Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and to maximize programmer productivity.” [7]

Adopting a design process based on an open source framework is one of the best approaches to addressing these challenges. These frameworks can be thought of as host platforms that allow stakeholders to focus on the mission, i.e. the added value of their product, rather than developing features that are common to all flight software. They strongly encourage the reuse of entire infrastructures, architectures, tools, and development processes in a particular application domain. Designing a framework for space flight software that can be used as a standard product for future space projects has become simpler and more relevant as space technologies have converged and standardized (e.g. LEON or ARM CPU, Spacewire [8] or Time-Triggered Ethernet [9] protocol, CCSDS SOIS EDS [10]).

Several actors, including government space agencies such as NASA and ESA, as well as private companies such as Bright Ascension [11] and Terma [12] , have developed mature frameworks, some of which have been flight tested and would save a significant amount of time and money on software projects. Most of these frameworks have been developed as part of internal projects and are intended to meet the specific needs identified by their use cases. Some of these frameworks have their source code available as open source on the Internet, allowing the entire domain community to audit the code, contribute to the project, and spread it to many international projects through capillary action.

Despite the study's emphasis on space flight software frameworks, they could be applied to any real-time or non-real-time embedded system. Their uniqueness in spaceflight stems from the fact that they were designed in an operational context with space domain-specific constraints. We will also find common hardware platforms and operating systems in this industry. Interacting with the embedded software is done via upward and downward flows of commands, telemetry messages, and events. Sector-specific international standards such as CCSDS or ECSS are also considered during the development cycle or when applying coding rules.

Although the study focuses on spaceflight software frameworks, they could be applied to any real-time or non-real-time embedded system. They are unique in the spaceflight domain since they were built in an operational context with space domain-specific limitations. In this area, we will also locate common hardware platforms and operating systems. Ascending streams of commands and descending streams of telemetry and event messages are used to communicate with the embedded software. During the development cycle or while implementing coding rules, international standards unique to the sector, such as CCSDS or ECSS, are also taken into account.

Because of their physical operating environment, spaceflight software systems are a distinct socio-technical system from embedded software systems. This physical environment forces their software engineers to consider other system components, how the software interacts with the hardware platform, and other system stakeholders, making it one of the most challenging software engineering systems.

Commercial CPUs, particularly those found in smartphones, bring performance and cost benefits that are piqueing the interest of the space community. However, as enticing as they may appear, their widespread use in space missions is still constrained. Radiation tolerance, power usage, and heat consumption are all examples of these constraints. The radiation environment in space has a direct negative impact on the hardware platform's components, and thus the software that runs on it. These space hardware platforms must incorporate components that implement specific rad-hard or rad-tolerant technologies, some of which are only available in the space industry and are highly costly. The disparity between space and consumer hardware platforms is exacerbated by these differences. Smartphones with several GB of flash memory or RAM and quad-core processors exist on the one hand, whereas space products with only a few MB of EEPROM or RAM and single or dual-core processors exist on the other.

This study first outlines a survey study on the same topic before presenting the context in which it was conducted. The essential guidelines of a flight software system architecture are then explained, along with their qualities attributes. The definition of what a framework is and should be aimed at is then followed by an evaluation of the three most promising frameworks currently available. Finally, the paper concludes with the most promising framework based on the aforementioned paradigm, discusses its implementation in an industrial setting, and provides a vision for their future.

2. RESEARCH CONFIGURATION

2.1. RELATED WORK

There are few comparisons of frameworks for spacecraft use available on the internet and easily accessible, on ResearchGate we found two studies detailed and relevant enough to be fully analyzed. Since the second study is based on the first, we will only report the results of the second study. This study report [13] entitled "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions", compares the following six Flight Software frameworks aimed at their reference nanosatellite mission VCUB1:

1. cFS;
2. SAVOIR-COrDET;
3. CAST;
4. GERICOS;
5. KubOS;
6. NanoSatMO.

In order to conduct their study this team compared six frameworks based on selection criteria. These selection criteria were chosen to address the specific issues of a New-Space mission as low cost, short schedule, management and reliability philosophy and use of COTS technology stacks. This results in the following:

1. Software code and documentation;
2. Flight heritage;
3. Small footprint;
4. Quality Attributes;
5. Long-term support;
6. Collaboration between users and the community;
7. The Consultative Committee for Space Data Systems (CCSDS) standardization.

Unlike their study, ours will focus on software architecture aspects for an AOCS/GNC platform equipment mission rather than a nanosatellite mission. Nevertheless, the specific problems of a New Space nanosatellite mission described in their study are applicable to ours.

Their study concludes that the NASA Goddard Space Flight Center cFS framework was the most balanced of all frameworks to meet their need. But the authors acknowledge that two other interesting frameworks not studied due to lack of time shall be considered in a future work: RODOS from DLR and F´ from NASA JPL.

2.2. RESEARCH BACKGROUND

This study was conducted over more than a year by a two-person team in parallel with our professional activities, between December 2020 and January 2022, and was divided into several major phases.

The first phase included a literature review of the current state of the art of software system architectures in general. In the first phase, we discovered that there were even more open source frameworks for flight software than we had anticipated. We were able to identify frameworks to implement in the second phase, as well as create the first drafts of an open source framework paradigm and scoring system. This was followed by a third phase of implementing an open source framework, during which we were able to see the limitations of our scoring system and expand the scope of a framework. Finally, a closing phase was carried out to ensure the consistency of the stated paradigm, the refinement of the scoring system and the veracity of the collected data.

We planned to implement the following frameworks just before the second phase:

- F´ (F Prime) from NASA Jet Propulsion Laboratory (JPL);
- cFS (core Flight System) from NASA Goddard Space Flight Center;
- SAVOIR-COrDET from ESA-ESTEC;
- FUHSI (Flexible and Unified Flight Software Architecture) from China Academy of Space Technology (CAST).

In order to get a better understanding of the frameworks, we decided to reduce the number of frameworks to be studied.

The FUHSI framework was removed from the list because, at the time of the search, it is not available as open source. Its developers announced at the Flight Software Workshop 2022 that they are considering making it available as open source. Its architecture is available in a CCSDS Orange book [14].

In discussions about our research with an ESA/ESTEC contact, he suggested we look at the SAVOIR-OSRA architecture designed for space applications. This is the ESA COrDET R&D activity that created the SAVOIR embedded software reference architecture (OSRA). Therefore, our contact advised us to look at their TASTE framework instead, as there is an ongoing project to align OSRA and TASTE to become one in the long run. Its recommendations were applied, and the COrDET framework was phased out in favor of the TASTE framework.

At the end of the second phase, the following frameworks were on the list to be studied:

1. F´ (F Prime) from NASA Jet Propulsion Laboratory;
2. cFS (core Flight System) from NASA Goddard Space Flight Center;
3. TASTE from ESA-ESTEC.

The implementation phase of the frameworks was carried out in an agile management style in order to be as efficient as possible and to study each framework in the same amount of time. Thus, this phase was seen as an epic composed of three releases, one for each framework. Each release was composed of five one-week sprints, with 15-minute meetings per day to follow up on the tasks.

2.3. RESEARCH METHODOLOGY

2.3.1. Mixed methods

This research is based on a mixed research methodology, where data collection methods are qualitative and quantitative, which forms the basis of this study report. This method was chosen because each of its two components provides the best result for this study. The relevance and rigor of this approach determine the adequacy of the results and the evidence they represent. Thus, a deficient methodology may lead to results that are interesting enough to lead to a continuation of the study, but erroneous results that do not bring any added value to what already exists.

2.3.1.1. Qualitative approach

The qualitative approach relies on impressions, opinions and views to collect data that describe frameworks and their context rather than measure them. This type of data is collected through:

- Interviews with key framework developers ensuring framework processes, maintenance, and evolution;
- Group discussions, which are the result of public or private exchanges that may take place with community members on community discussion forums (such as the Discussions feature of GitHub, the developer social network);
- Observations, which result from the user experience the research team had while studying a framework;
- Documentation that results from reading international standards or research articles applicable to the field of study.

Admittedly, this method does not provide precise data for important issues and may lack rigor or rationality, but it does allow us to integrate a human factor. This human factor can help us in the final phase of the study and guide us in the follow-up of the current study. The answers obtained from the open-ended questions can add a human touch to the purely quantitative data of our results and can help us to better understand unknown or even unsuspected aspects.

2.3.1.2. Quantitative approach

The quantitative technique will use a criteria-based evaluation system to objectively examine and judge the quality of each framework. This evaluation system is made up of a set of quality attributes that are applicable to software frameworks and will be discussed further in this document. The scores may not reflect the true quality of each framework due to the empirical experience of each member of the jury; however, these scores will reflect the apparent quality of each framework for a non-initiated user who discovers them. The study's evaluation system can be used as a model for other more in-depth studies or frameworks, in whole or in part.

2.3.2. Data collection

The investigation technique used to collect relevant data is essential because it is the relevance of the data collected and its proper interpretation during the analysis and conclusion that will bring interesting results to the study.

Since the field of study is software engineering, the data collection will take the form of a software Proof-Of-Concept (PoC). This Proof-Of-Concept will be confronted in its architecture with various software engineering problems. Among these problems we will find for example:

- The planning of an algorithm at a certain periodicity;
- The logging of events related to the execution of the program;
- The restitution of error code or telemetry messages;
- The use of third-party software in the form of software libraries (compiled external code).

Data will be collected through the implementation of the same Proof-Of-Concept for each framework. The development cycle will explore training resources and technical documentation; gain knowledge of the framework architecture, development support tools, test tools, ground support system. It will also provide an opportunity to compare the overall benefits and drawbacks of the frameworks.

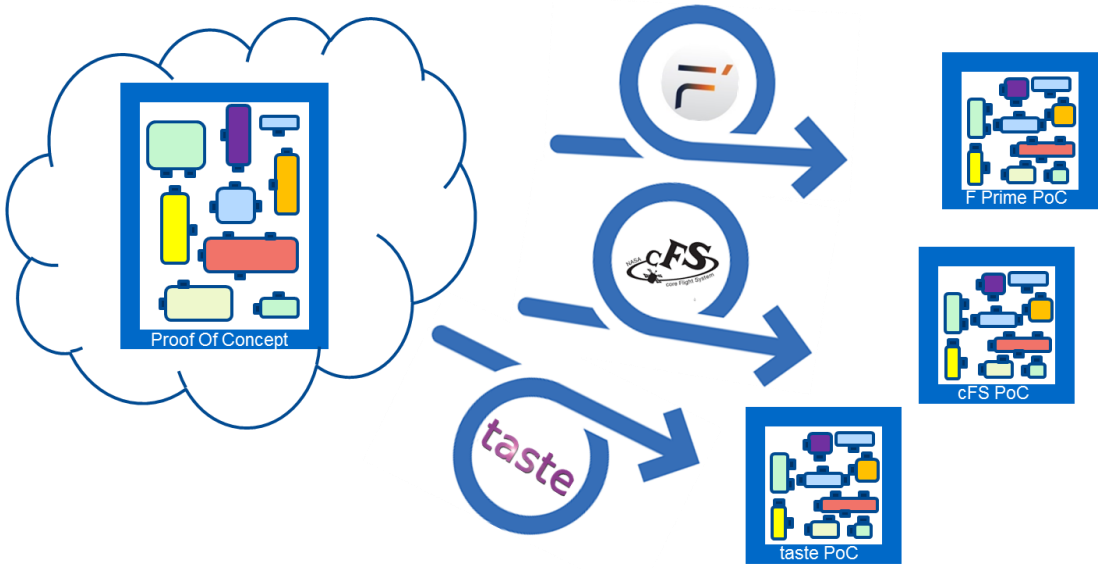


Figure 2-1: Overview of data collection technique

Section 3 will go over the specifics of the Proof-Of-Concept implementation.

2.3.3. Data Analysis

Once the empirical data have been collected, we can rate each framework against each criterion. The scoring of each criterion was the result of mutual agreement among the people who implemented the frameworks during the study. In practice, this scoring was repeated several times in order to avoid being biased by hasty conclusions, to unconsciously create a rating scale, and to produce scores that more rationally reflect our final overall feeling.

3. PROOF-OF-CONCEPT TOPOLOGY

The Proof-Of-Concept system topology is broken down into discrete components with well-defined interfaces. The level of granularity has been kept relatively low since it is only a mock-up. Thus, there are six components/applications each encapsulating a specific service that could be common to other missions in different operational contexts.

The inter-component interfaces allowing data exchanges between one and several components are of two types: commands and reports. Commands are sent by a user service to trigger actions and transmit the necessary data to the service provider. As for reports, they are returning of functions, telemetry messages or error code messages that the service provider provides to the user of the service.

We have not taken the liberty of imposing that these inter-component interfaces shall conform to a standard format such as CCSDS SOIS EDS or CCSDS Space Packet Protocol (SPP). This is because some frameworks may not implement these standards by default and the effort to use them can significantly slow down the development of the mockup.

3.1. PROOF-OF-CONCEPT ARCHITECTURE

For this Proof-Of-Concept, we wanted to replicate a basic data flow of a space camera (star tracker, rendezvous camera, monitoring camera…) using three commands and telemetry messages:

- Photo Cmd/Tlm;
- Engineering Cmd/Tlm;
- Housekeeping Cmd/Tlm.

We divided the software system into six (6) components with afferent (incoming connections) and efferent (outgoing connections) couplings to handle these different commands:

1. **OHTempAcq**: in charge of generating the temperature of three (3) Optical Heads (OH)
    a. The temperature intervals are respectively [0,30], [30,60], [60,90] for each Optical Head;
2. **OHAcq**: in charge of generating an image according to the temperature of the OH;
3. **ImgProc**: in charge of the background calculation of the image;
4. **PosProc**: in charge of calculating a new position/orientation with an external quaternion library;
5. **Housekeeping**: in charge of retrieving system information about the state of the system.
6. **Time**: in charge for generating an internal clock that counts the number of cycles performed by the program, allowing clocking other components.

We can see on the diagram below which components are linked together and their role:

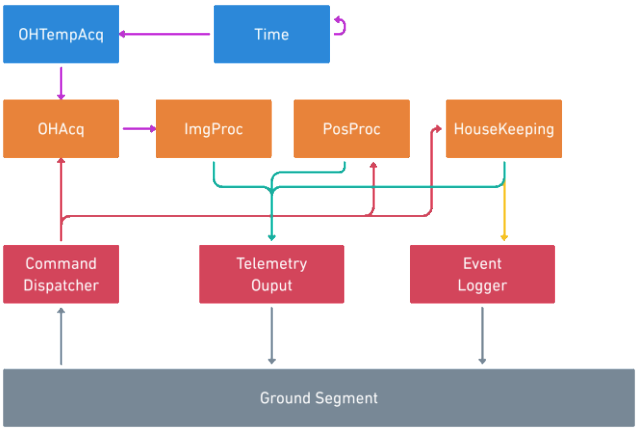


Figure 3-1: Topology of the PoC system

Each time a command is sent, the component “Command Dispatcher” is in charge of dispatching the request to the right path in order to perform the expected actions.

3.2. SYSTEM BEHAVIOR

3.2.1. Photo Command and Telemetry

Here is a description of the Photo command and telemetry message:

Photo command (GDS ->FSW)					
Field name	**Field description**		**Range**	**Unit**	**Format**
CmdId	Command identifier		1	-	U32
RoI_Width	Region of Interest parameter	Number of column of the RoI	0..2048	pixel	U16
RoI_Height		Number of lines of the RoI	0..2048	pixel	U16
RoI_OffsetX		Offset of the first column of the RoI	0..2048	pixel	U16
RoI_OffsetY		Offset of the first line of the RoI	0..2048	pixel	U16
OH_Id	Optical Head identifier		1..3	-	U16

Figure 3-3: Photo command data format

Photo telemetry (FSW -> GDS)				
Field name	**Field description**	**Range**	**Unit**	**Format**
TlmId	Telemetry identifier	1	-	E32
BkgndSig	Background signal	-	-	U64
StdBkgndSig	Standard deviation of background signal	-	-	U64
OH_Id	Optical Head identifier	1..3	-	U16
Timestamp	Timestamp of the image	-	ms	U64
Temp	Temperature associated with the image	-	°C	U64

Figure 3-2: Photo telemetry data format

In our PoC when receiving this command, three components will be implied in the process: **OHTempAcq**, **OHAcq** and **ImgProc**.

The **OHTempAcq** component will first send to **OHAcq** component the three temperatures generated at an interval T, by the **Time** component, for each Optical head.
Then **OHAcq** will generate an image based on the temperature of the associated optical head. Once the image is generated, the component will send only the Region Of Interest (ROI) of it to the **ImgProc** component. The **ImgProc** component will then calculate the background signal and its standard deviation of the image and then return these values to the **TelemetryOutput** component which will send the corresponding telemetry message.

3.2.2. Engineering Command and Telemetry

Here is a description of the Engineering command and telemetry messages:

Engineering command (GDS ->FSW)					
Field name	**Field description**		**Range**	**Unit**	**Format**
CmdId	Command identifier		10	-	U32
Org_X	Start Position (3D vector)		-	-	U64
Org_Y			-	-	U64
Org_Z			-	-	U64
Org_X	Position Translation (3D vector)		-	-	U64
Org_Y			-	-	U64
Org_Z			-	-	U64
RotAngl_X	Rotation angle (quaternion)	Scalar part of the quaternion	-	-	U64
RotAngl_Y		Imaginary i part of quaternion	-	-	U64
RotAngl_Z		Imaginary j part of quaternion	-	-	U64
RotAngl_T		Imaginary k part of quaternion	-	-	U64
RotAxis_X	Rotation Axis (quaternion)	Scalar part of the quaternion	-	-	U64
RotAxis_Y		Imaginary i part of quaternion	-	-	U64
RotAxis_Z		Imaginary j part of quaternion	-	-	U64
RotAxis_T		Imaginary k part of quaternion	-	-	U64

Figure 3-5: Engineering command data format

Engineering telemetry (FSW -> GDS)					
Field name	**Field description**		**Range**	**Unit**	**Format**
TlmId	Telemetry identifier		10	-	U32
CurrPos_X	Current position (3D vector)		-	-	U64
CurrPos_Y			-	-	U64
CurrPos_Z			-	-	U64
CurrAtt_X	Current attitude (quaternion)	Scalar part of the quaternion	-	-	U64
CurrAtt_Y		Imaginary i part of quaternion	-	-	U64
CurrAtt_Z		Imaginary j part of quaternion	-	-	U64
CurrAtt_T		Imaginary k part of quaternion	-	-	U64

Figure 3-4: Engineering telemetry data format

When we receive an **Engineering** command, only **PosProc** component will be used. **PosProc** component will use the parameter contained in the command to calculate the new position and a quaternion associated to the Space System. Once this is done it sends all these measurements to the **TelemetryOutput** component which will send the corresponding telemetry message to the **Ground System**.

3.2.3. Housekeeping Command and Telemetry

Here is a description of the Housekeeping command and telemetry messages:

Housekeeping command (GDS ->FSW)				
Field name	**Field description**	**Range**	**Unit**	**Format**
CmdId	Command identifier	20	-	U32

Figure 3-7: Housekeeping command data format

Housekeeping telemetry (FSW -> GDS)					
Field name	**Field description**		**Range**	**Unit**	**Format**
TlmId	Telemetry identifier		20	-	U32
SysMode	System operational mode		-	-	U64
OH1Temp	Temperature of the first Optical Head		-	°C	U64
OH2Temp	Temperature of the second Optical		-	°C	U64
OH3Temp	Temperature of the third Optical Head		-	°C	U64
major_ver	Software version	major	-	-	U16
minor_ver		minor	-	-	U16
patch_ver		patch	-	-	U16
SysCurrDate	System current date		-	ms	U64

Figure 3-6: Housekeeping telemetry data format

The Housekeeping command simply sends a request message to the system, which then activates the Housekeeping component to return the Housekeeping telemetry message. This telemetry message provides the system user with operationally important information, i.e. maintenance and diagnostic information that is not provided in other telemetry messages.

4. SPACE FLIGHT SOFTWARE ARCHITECTURE GUIDELINES

Most of the time, when we talk about software architecture, we refer to a loosely defined notion of the most important aspects of the internal design of a software system [15]. This notion is often decoupled from its programming by computer languages.

However, one way to define software architecture, according to Ralph Johnson [16], is the developers' shared understanding of the system design [17].

According to Martin Fowler [18], proper software architecture is important; otherwise, adding new features requested by stakeholders becomes more difficult and expensive [19]. Indeed, poor architecture contributes significantly to the growth of garbage, which is software that prevents developers from understanding the source code. Because there are more obstacles to overcome, software with a lot of this garbage is significantly more difficult to modify and maintain, and it leads to slower feature additions with more defects.

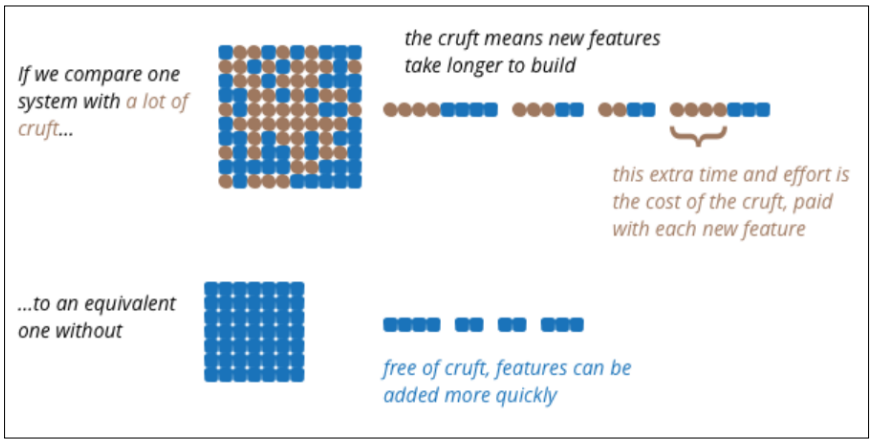


Figure 4-2: Illustration of the impact of garbage (cruft) on feature development. - © Martin Fowler

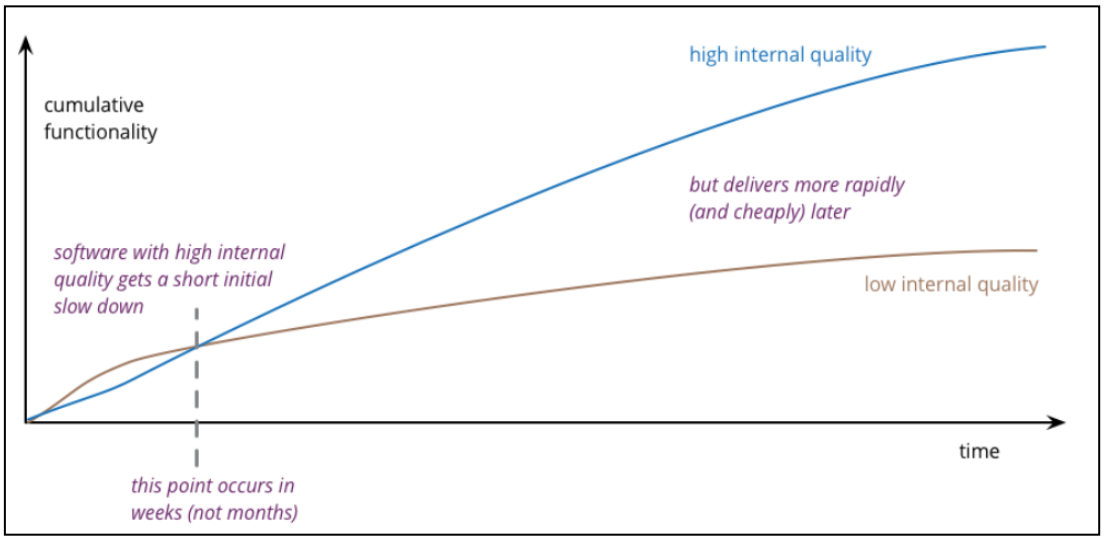


Figure 4-1: Impact of internal quality on feature development - © Martin Fowler

Before stating a framework's paradigm, it is necessary to comprehend where it stands in the minds of its users. And, because failing to follow best practices can result in significant technical debt, we will use Uncle Bob's Clean Architecture pattern [20] as a reference in the field of software system design for this. It is similar to other design patterns such as the hexagonal architecture [21] of Alistair Cockburn [22], the Onion architecture [23] of Jeffrey Palermo [24] or the micro services architecture.

Uncle Bob's architecture schematizes a software system as a series of concentric circles representing the various domains of a software and is based on the inversion of dependencies derived from the SOLID principles [25]. The inner circles in this diagram can known everything about an outer circle's software entities (functions, classes, variables, etc.). The code of an outer circle must not refer to the software entity declared in an inner circle. In other words, dependencies in source code should only point inward, to higher-level policies specific to the business logic.

Entities are the rules that define the company's business domain; these are critical rules that implement the business logic that can be reused in multiple missions and are at the heart of the system. These rules represent the system's most valuable contribution to the market in which it operates. An industrialist will regard these rules as intellectual property and will take steps to protect, preserve, and perpetuate them. In theory, the implementation of this domain's logic should not rely on third-party technologies, i.e. it should be isolated to limit the impact of technological evolution. Indeed, external technologies may evolve, but this is not always a business requirement, which is why the idea of isolating the business domain in order to perpetuate it exists. It is recommended to use the Domain-Driven Design approach [26] introduced by Eric Evans [27] in 2003 to differentiate the business logic from the system as a whole.

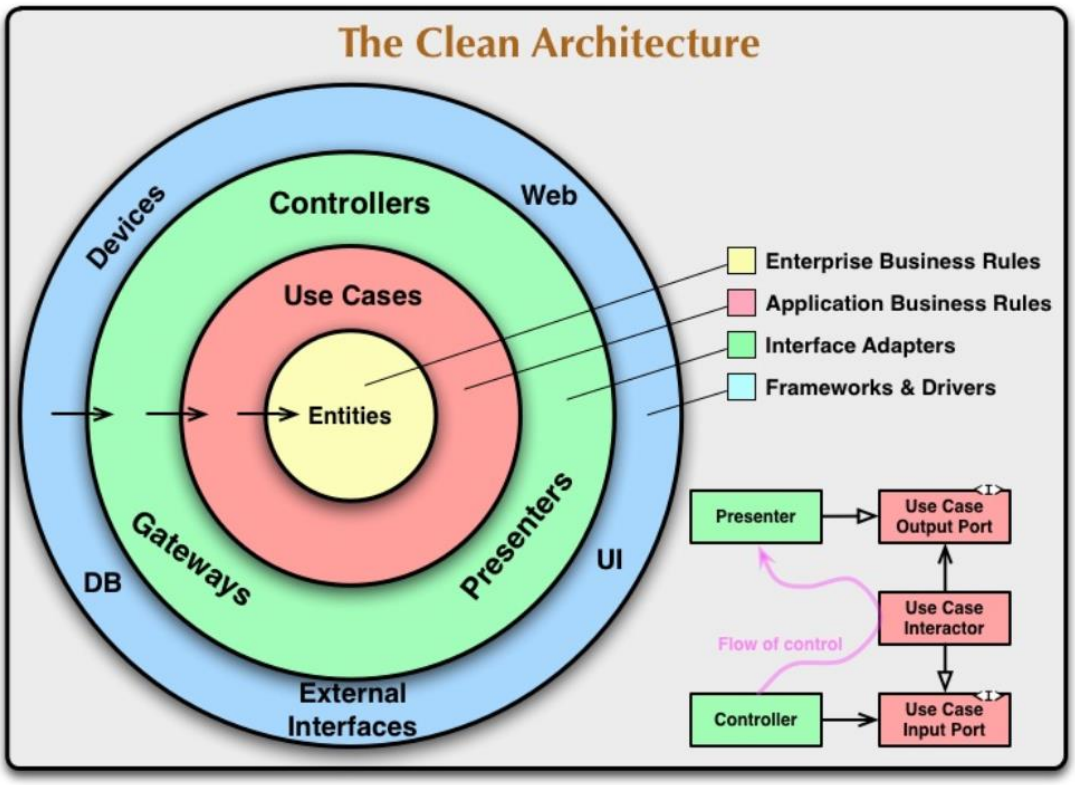


Figure 4-3: Uncle Bob's clean architecture illustration - © Robert C. Martin

Let's look at a star tracker [28] as an example. This instrument, which is part of a satellite's AOCS/GNC system, is used for spacecraft navigation and continuously monitors the stars to ensure that the spacecraft is always on course and pointed in the right direction. The star tracker takes a photograph of the star configuration in its field of view and compares the image, i.e. the observed star configuration, to its built-in star maps. In this case, the scope of work will include everything related to image processing algorithms as well as the parameterization and calibration of optical heads.

The Use Cases that follow contain the mission-specific rules that encapsulate all of the system's use cases. This layer orchestrates data flows from and to entities, defining how the system user and business logic interact.

The architecture of the software system revolves around these use cases. In the case of a star tracker, this could correspond to the instrument's operating modes, such as standby mode, acquisition mode, which photographs the sky in search of known stars, or tracking mode, which follows the stars in order to provide the spacecraft's attitude over time.

The Interface Adapters layer is a collection of adapters that convert data from the format that is most convenient for use cases and business logic to the format that is most convenient for specific technology stacks, such as an external controller or a database. Thus, it is partly an abstraction layer that enables the Use Case layer to be agnostic to the underlying technologies. In the case of a space product, we might find different stacks of communication or compression protocols, such as those of the CCSDS SOIS protocols or the CCSDS Space Packet Protocol, in these layers.

Finally, we come to everything related to the infrastructure and environment in which the software system was developed or deployed. A well-designed architecture is free of the technologies implemented in this layer, which can evolve over time and whose use can be postponed and deferred. This layer is of particular interest in our research because it contains frameworks. We understand that frameworks are placed in the outermost layer, as far away from the high-level rules determined by business rules as possible. Therefore, frameworks are details that should not impose constraints when defining the architecture of a software system.

However, not all stakeholders are capable of successfully integrating this type of architecture into their systems.

Some stakeholders, such as startups or new industrial projects/programs with limited budgets, will not have the time to implement this architecture, which may appear complex at first glance, and will instead rely solely on the framework.

Indeed, these players are under pressure to deliver a marketable product, please their venture capitalists, and complete their missions. That is why we think it is essential to recall some key guidelines that framework publishers should keep in mind when developing their frameworks.

4.1. COMPONENT-BASED SOFTWARE ARCHITECTURE

The software architecture pattern that most often appears in the literature to promote the reuse of software components on different missions/projects is the component-based architecture, whether it is a purely modular monolith, service-oriented architecture or micro services architecture [29] [30]. That architecture is gradually supplanting traditional monolithic architecture as the primary design principle for new applications in both centralized and distributed systems [31][32].

The component-based architecture pattern results from the desire to build systems by assembling compliant 'plug-in' components in the same way that certain things are done in the physical world. In practice, achieving the promise of "plug-in" or "LEGO" type software development is extremely difficult. "Software reuse is more like an organ transplant than snapping together Lego blocks," says John D. Cook [33]. One of the drawbacks of all reusable components is that the more effort that is put into making them reusable, the harder they are to use.
Improving the degree of reusability of a component involves adding additional hooks, options, and decision points to accommodate different use cases, which detracts from the usability of the component.

This architecture pattern consists in discretizing a software system to favor a finer granularity of reuse of its elements, to gain in modularity, and so that its evolution requires only a minor and isolated modification of the source code.

The granularity of services chosen as the basis for service identification is crucial. There is insufficient flexibility in the potential deployment of system compliant with the component-based architecture at too high a level of granularity, too few services, and few of the benefits of the component-based architecture will be realized. When the level of granularity is too low, the component architecture becomes overly prescriptive; a given function is presented with too many potential interfaces, ultimately reducing the scope of compatible 'plug-in' components; and the size of the standardization task becomes unmanageably large.
Identification of the fundamental types of information exposed at end-to-end inter-functional interfaces at the selected level of granularity is a critical step in service identification.

The software system is decomposed into discrete elements whose interfaces are described with well-defined semantics. These elements, called components or applications, are software units, a set of logically and functionally related capabilities that a component offers to other components and that are specified in terms of:

- Services it offers to other components.
- Services it needs from other components.

The service concept sees a component as a provider of services to other components and as a user of services from other components.

The modularity of the software is expressed by the services rendered by its components that can be replaced, reused or upgraded and deployed independently. But this modularity is also achieved through the use of libraries that are part of a service. While keeping in mind that these libraries should not represent the whole software but a service. Of course, a component-oriented architecture uses libraries, but the key is to decompose the software system into components and to decompose it into services. Libraries are defined as elements linked to a component and called by means of function calls. While components are called by various mechanisms such as a service bus, communication protocols, etc.

The benefit of breaking down the software system into components is that they can be deployed separately in different partitions within the same hardware platform, with processors optimized for specific domains.

Indeed, some hardware platforms include SoCs with a variety of processing units, most of which are multicore in design, and some of which focus on real-time, security, or data processing issues. These

platforms are becoming more popular as they promote lighter, smaller, cheaper, higher performing, and more reliable space electronics [34].

In the past, we worked on a technology demonstrator for a new space application that was to run on a Zynq UltraScale+MPSoC. This SoC is equipped with two processing units: a real-time processing unit (RPU) based on an ARM Cortex-R5 processor and an application processing unit (APU) based on a high-performance 64-bit ARM Cortex-A53 processor. A non-exhaustive representation of the environment is shown below.

This configuration offloads the real-time components to the RPU, freeing up all of the APU's power for the data processing components. In contrast to a monolithic architecture, a CBSE design allows the full capacity of these hardware platforms to be utilized more efficiently. If the software system had a purely monolithic design, it would be unable to take advantage of the full capacity of this type of SoC. These monolithic architectures, let alone modular monolithics, have only one binary and are difficult to partition by isolating components by service.

This approach fosters more innovation, allows specialization and differentiation of components and ensures rapid integration into a system [32].

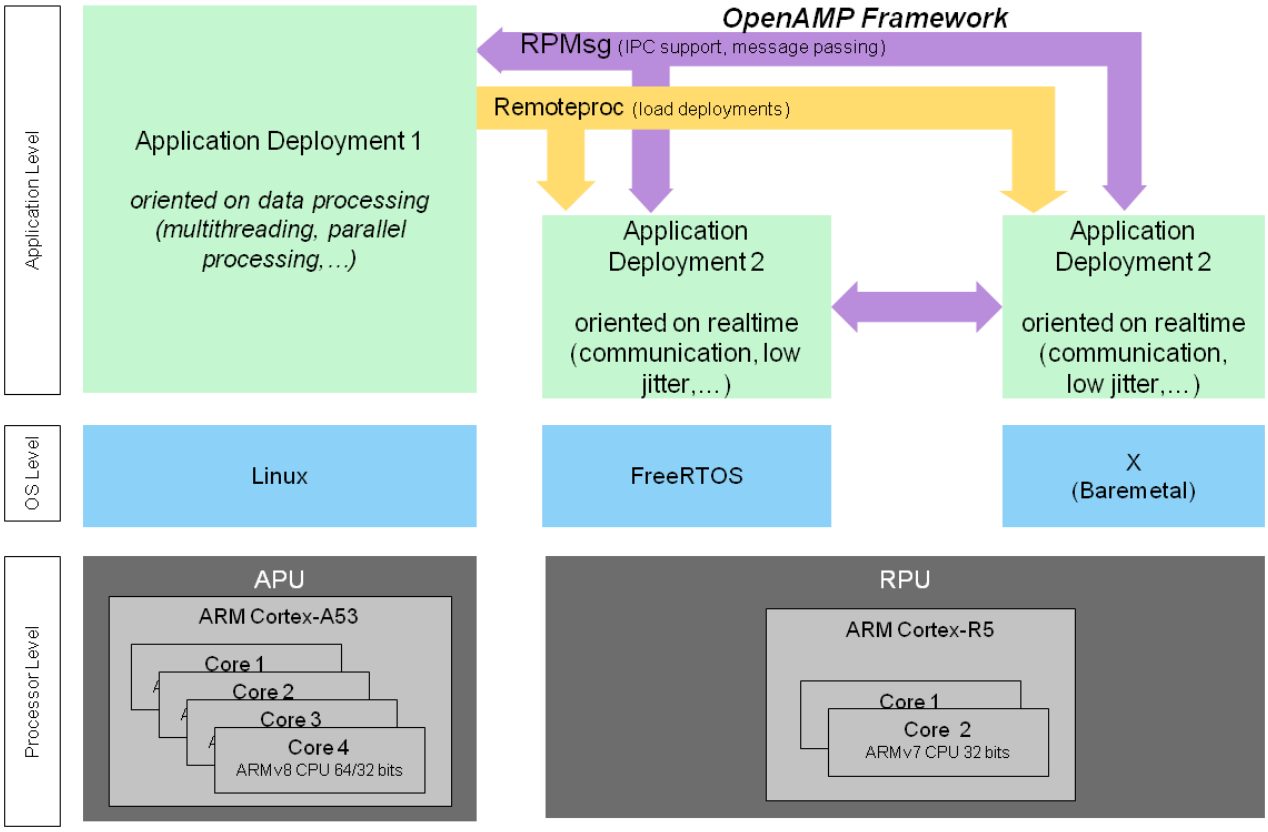


Figure 4-4: Heterogeneous deployments in Zynq UltraScale+MPSoC

4.2. ABSTRACTION RATHER THAN CONCRETION

Abstraction layers are critical programming components in software engineering. They strive to avoid concreteness at key boundaries in a software system's architecture in order to improve reusability, testability, interoperability and adaptability [35]. They create a new semantic level by unifying the interfaces of components with similar functionalities in a coherent manner [36].

This new semantic treats its components as details and allows its user to be agnostic by not revealing the implementation and underlying complexity. Operating systems (OS), board support packages (BSP), communication protocols, and databases are examples of these components. Each of these components can be implemented using technologies that must be kept as far away from the system's overall business goal as possible. Without these abstractions, the software system contains far too many concretions involving close coupling with technologies that may become obsolete over time.

Operating System Abstraction Layers (OSAL) or Hardware Abstraction Layers (HAL) are typically observed in embedded software systems.

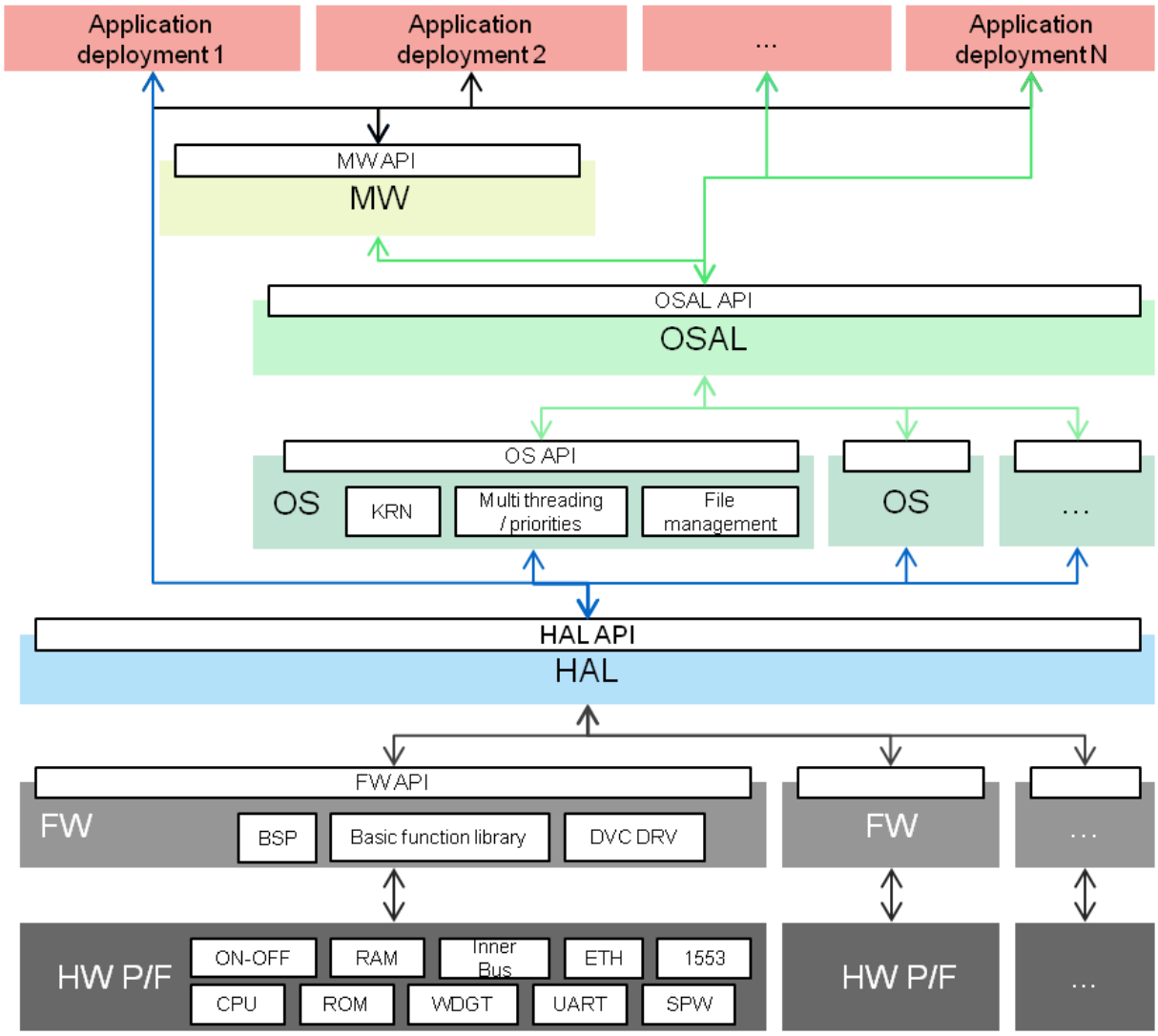


Figure 4-5: OSAL & HAL illustration

OSAL

An OSAL is a collection of programming interfaces for a fictitious operating system that can perform all of the operations of a real operating system. It shows the user an abstraction of the common functionality provided by all real-time operating systems (RTEMS, VxWorks, Azure ThreadX, QNX, FreeRTOS,...) or not (Linux, WinCE,...). We find the following services:

1. multitasking management with prioritization ;
2. synchronization;
3. file management;
4. message queues;
5. communication; and
6. timeout.

OSAL provides each of these services by wrapping the concrete entities of the operating systems it encapsulates in thin wrappers.

This layer enables the development and testing of adaptable software systems that are not dependent on any particular operating system, allowing development to take place on the development station [37]. It reduces the time and effort required for developers to migrate a software system from one operating system to another.

HAL

Hardware abstraction layers (HALs) are commonly integrated into operating systems or used at the same level as an OSAL to prevent the software system from becoming overly reliant on the hardware platform [38]. In a baremetal system, for example, a HAL will suffice to abstract the software system from its hardware environment. By providing a consistent interface, HALs allow applications to interact with a fictitious hardware platform. Through this interface, the software system has direct access to hardware devices such as input/output ports, memory, or interrupts.

These HALs include boot loaders, board support packages, device drivers and other components. The system can run on a variety of hardware platforms as long as their BSP is integrated into the HAL. This layer is particularly relevant in the context of embedded systems because the hardware architecture can vary from project to project.

This layer is also important because each hardware platform may have a distinct build chain. In the case of a build chain for a C program, it imposes concretions on the user: on the program source code and on the configuration of the build chain.

The supplied header files contain programming objects that are not necessarily standard to the C programming language and are interpreted by the vendor's build chain to allow manipulation of the material.

On the one hand, some objects may be given compiler and linker-specific attributes. The "weak" label given to functions that benefit from polymorphism is especially noticeable.

BSP developers provide function declarations whose logic is supposed to be implemented by the user. These functions are called in the BSP code at strategic points such as startup, error cases and logs. The body of these functions is empty by default, but by including the "weak" attribute, the compiler will replace the default definition with the user's definition, if it exists.

A code snippet from Microchip's SAMRH71 evaluation kit's BSP [39] demonstrating the use of polymorphism via the "weak" attribute for a mock definition of interrupt vectors is provided below.

```
extern void Dummy_App_Func(void);

/* Brief default application function used as a weak reference */
void __attribute__((optimize("-O1"),long_call))Dummy_App_Func(void)
{
    return;
}

/* Optional application-provided functions */
extern void __attribute__((weak,long_call, alias("Dummy_App_Func"))) _on_reset(void);
extern void __attribute__((weak,long_call, alias("Dummy_App_Func"))) _on_bootstrap(void);

/**
 * \brief This is the code that gets called on processor reset. To initialize the device, and call the main() routine
.
 */
void __attribute__((optimize("-O1"), section(".text.Reset_Handler"), long_call, noreturn)) Reset_Handler(void)
{
    /* ... */

    /* Call the optional application-provided _on_reset() function. */
    _on_reset();

    /* ... */

   /* Call the optional application-provided _on_bootstrap() function. */
    _on_bootstrap();

    /* ... */

    /* Branch to application's main function */
    int retval = main();
    (void)retval;

    /* ... */
}
```

Some objects, on the other hand, give the user access to processor functionality, such as writing/reading processor registers, accessing I/O ports, and the interrupt table. However, if the BSP API is neat enough to keep an architectural boundary with the user code, this disadvantage is less severe than the previous one.

Another BSP code snippet from Microchip's SAMRH71 evaluation kit's BSP [40] is shown below, demonstrating the processor's floating point instruction activation using exotic symbols.

```
/* Enable FPU */
__STATIC_INLINE void FPU_Enable(void)
{
uint32_t prim;
    prim = __get_PRIMASK();
    __disable_irq();

     SCB->CPACR |= (0xFu << 20);
    __DSB();
    __ISB();

    if (!prim)
    {
        __enable_irq();
    }
}
```

Therefore, the user has source files that are more or less tightly coupled to this specialized build chain.

The second concretion related to build chain configuration stems from the fact that the compiler and linker may have options that are unique to them and may be bewildering to other compilers. These options can be specified in the build chain's configuration files (CMake file, Makefile, etc...), as well as in the source code. For example, on an ARM processor, we will use the options "*-mfloat-abi=hard*" or "*-mfloat-abi=softfp*" to specify whether our application uses hardware instructions or software library functions for floating point operations.

4.3. SOFTWARE SERVICE BUS

A direct connection between services is not a sustainable or easy to maintain solution because it creates a strong coupling between them. Changing the location of a service in the topology necessitates the updating of all connections to that service.

To achieve software modularity, it is therefore best to have a hub in the form of a service bus, analogous to computer buses such as PCI [41]. The software bus connects software components in the same way that the PCI bus connects hardware components such as GPUs, MIL-STD-1553 cards or SpaceWire cards. This hub service acts as a mediator and provides a solution to the problem of complex event interaction. It makes the system much easier to maintain than a single end-to-end service connection. It supports integration tasks such as message transformation: by defining a standard protocol for application data exchange, it enables the interconnection of all components that adhere to it.

This service can either follow a standard defined within the framework of a mission for its internal operation, or follow an international standard such as the Space Packet Protocol [42]. In terms of its implementation, this service takes the form of a component in its own right in the topology of the software system and constitutes the bus itself. The communication via this bus can be done by a design pattern Publish/Subscribe (PubSub) and be realized with the help of a message oriented middleware with message queues.

The role of this service is to route messages between services, monitor and control message exchange, resolve conflicts between components, and provide such facilities as event processing and message queuing.

This service offers the following benefits:

- a more plug-and-play approach.
- better scalability of services;
- workload distribution among different processing units;
- an easier way to design loosely coupled services;
- services that can be replaced.

The disadvantages, on the other hand, are mainly focused on the following:

- A single point of failure: if the hub fails, the system also fails;
- Message queuing, message translation and other common services can degrade performance, for example by adding overhead, or even becoming a bottleneck.

The diagram below is from the cFS "Background and Overview" material [43], and it depicts the transition from tightly coupled components with specific interfaces to loosely coupled components with standardized interfaces. The software bus makes it easier to create, add, delete, reconfigure, and redistribute components over time [44].

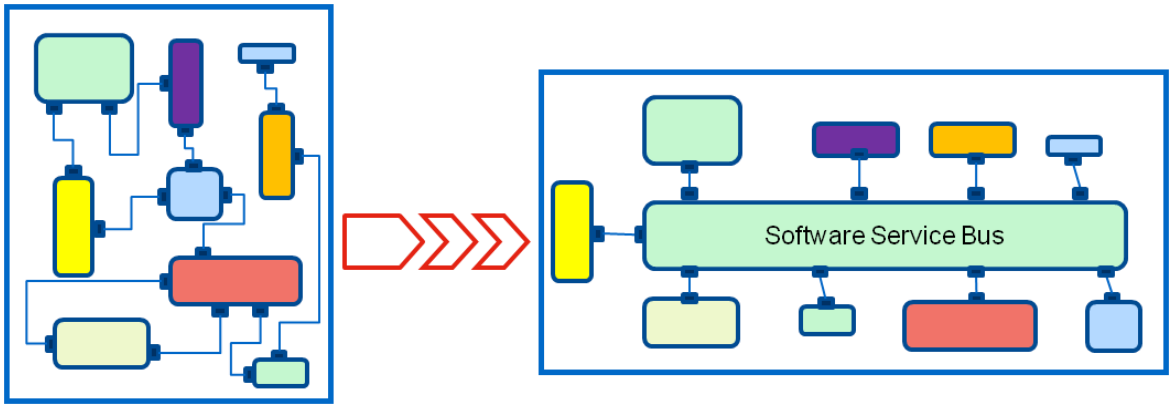


Figure 4-6: Paradigm shifted by the Software Service Bus

However, as indicated in the book "Software Architecture: The Hard Parts" [45], the choice between orchestration (central coordination) and choreography (no central coordination) is a compromise.

On the one hand, the Software Service Bus allows to:

- reduce system complexity;
- simplify error management;
- improve recovery capacity;
- and simplify state management.

On the other hand, it:

- Reduces responsiveness because all communications pass through it, which can lead to a throughput bottleneck.
- Reduces fault tolerance because it introduces a potential single point of failure into the system, which can be resolved by redundancy but adds complexity.
- Reduces scalability by reducing potential parallelism.
- Increases service coupling because domain components may need to be coupled to it when they should be infrastructure independent.

4.4. DECENTRALIZED DATA MANAGEMENT

Each service should have its own database and should not share the same database with other components [46]. Indeed, the separation of databases promotes isolation because the teams developing the service know that the service has its own database and that there is no access to data stored in other services. The service design will become more isolated, less coupled/dependent. In this way, we should end up with a more modular system with more autonomous services that can be modified without any effect on other services.

Furthermore, according to the book "Software Architecture: The Hard Parts" [45], data separation eliminates a single point of failure in the system, thus providing better fault tolerance. In other words, it ensures that some services/components remain operational in case of database failure.

The separation of the database should be decided based on a case study with trade-offs, as it may result in duplication of data that we cannot afford in an embedded system with limited memory resources.

5. SPACE FLIGHT SOFTWARE ARCHITECTURE QUALITY ATTRIBUTES

Quality attributes are a set of functional and non-functional requirements of a system, i.e. the explicit and implicit needs of its various stakeholders. They form the basis of a system for evaluating the overall quality of a software system [47].

High scores in the software quality attributes allow software architects to ensure that a software application will perform according to the specifications provided by the customer, and will only accumulate a suitable amount of waste over time that will not impact the life cycle of a software system.

Most often, these elements are not considered at the beginning in small projects, small companies or start-ups. Indeed, the cost of maintaining quality attributes is high and the result is only visible on a large scale.

To take them into account during the development of a project, the software architect, along with the development team, shall lobby the project to consider them. Because even if they are not directly beneficial to the project, they will be beneficial to the product line in which the project fits or even to the program.

However, as development teams and products grow, abandoning these quality attributes leads to increased technical debt and quality problems. Addressing quality attributes of a software system improves the Return on Investment (RoI) of software products and also produces higher quality products [48].

Depending on the type of software system being considered, some of these attributes are more important than others. In a perfect world a software system should meet all the quality attributes that are important to stakeholders, but this is a very, if not too, complicated task. Finding the right set of quality attributes is the first step in quality control and management. Achieving high quality requires measuring, controlling, managing and improving quality [49].

At the beginning of the study, we simply wanted to continue and improve the work of [13], i.e., to improve the selection criteria and focus on fewer spaceflight software frameworks for further investigation. As the study progressed, the list of selection criteria grew significantly, we exceeded forty criteria, and some were complicated to evaluate because they did not apply to every framework studied. Adding a weighting to each criterion complicated the overall scoring and the maintenance burden was too great for the research team.

We decided to discard the list of selection criteria and replace it with a more standardized and generic method of judging software quality, i.e., evaluating frameworks against the quality attributes applied to Space Flight software.

In May 2016, at a conference [50] hosted by the Software Engineering Institute, NASA's Software Architecture Review Board presented a beginning list of key quality attributes applicable to spaceflight software that are embedded, real-time, mission-critical software systems [51]. The presented list contains at least 14 key quality attributes, but is intended to be expanded over time.

We ended up choosing to re-use the ISO/IEC-25010 product quality model, which segregates software product quality into eight core quality attributes. We have incorporated the fourteen core quality attributes from the NASA Software Architecture Review Board into this model, along with a more detailed description of our understanding of these attributes, and we will also include attributes that we believe should be considered in these frameworks because they do not uniquely encompass space flight software.

Figure 5-1: Product quality model defined in ISO/IEC 25010

Some attributes are not straightforward to evaluate rationally or do not seem applicable in whole or in part to the frameworks at first glance, so we will let them appear in order to keep track of our thoughts and to give possible ideas for future research on the same area.

5.1. FUNCTIONAL SUITABILITY

Functional Suitability is defined in ISO/IEC 25010 as the degree to which a product or system provides a function that meets explicit or implicit requirements, i.e. functional and non-functional, when used in specific operating contexts, in this case a mission [52].

Since it is complicated to know the non-functional requirements of a software system, this attribute can be evaluated based on the following sub-characteristics:

- **Functional completeness** is the ability of a software system to cover all the requirements specified by the product's stakeholders.
- **Functional correctness** is the ability of a software system to provide correct measurements with the required level of accuracy.
- **Functional appropriateness** is the ability of a software system to respond to different specified operating contexts.

As we can see, these attributes apply more to the user's use of the framework than to the framework itself. Of course, a framework should allow the user to fully control whether these requirements are met; however, evaluating these requirements at the framework level seems too difficult because it also depends on the operating system, hardware platform, etc. Therefore, these features will be excluded from the evaluation system.

5.2. PERFORMANCE EFFICIENCY

Performance Efficiency is defined in ISO/IEC 25010 as the ability of a software system to comply with temporal requirements to complete a given task, or the efficient use of hardware platform resources under given conditions.

From the point of view of temporal requirements, this usually involves verifying in functional tests that the software system responds to various events within defined time frames. These events can be clock events, process interrupts, commands.

From the point of view of hardware resources, the software system must make efficient use of the processor capacity, volatile and non-volatile memory space.

This attribute is evaluated on the basis of the following sub-characteristics:

- **Temporal Behavior** is the ability of a software system to meet requirements on response time, processing time and transmission time of ascending and descending data flows.
- **Resource Utilization** is the ability of a software system to meet requirements on the quantity and type of resources used during its execution.

- **Capacity** is the ability of the maximum limits of the parameters of a software system to meet the requirements.

Performance Efficiency, like Functional Suitability, will not be included in the final scoring system for the same rationale.

5.3. COMPATIBILITY

Compatibility represents the ability of a software system to operate satisfactorily on the same hardware platform without affecting the other software systems with which it shares this platform, in the case of a centralized architecture.
But it is also the capacity of a software system to exchange information with other systems without conflict problems, which is the case of a distributed architecture. This attribute is evaluated on the basis of the following sub-characteristics.

5.3.1. Coexistence

Coexistence is the ability of a software system to function nominally in conjunction with other software systems on a hardware or software platform without negative impact on the platform [53]. In this study, we will ask whether a software system developed from one framework can be integrated or interfaced with another software system that was developed with another framework.

5.3.2. Interoperability

Interoperability is the ability of a software system to exchange data with other software systems via known software interfaces.

In the current context and in our domain, the use of the international standard CCSDS SOIS EDS, which aims to homogenize the data exchange format in space systems, results in a high degree of interoperability. Conversely, frameworks incorporating a proprietary data serialization format result in a low level of interoperability. In this case, these frameworks must be flexible enough to allow stakeholders to use standards such as CCSDS SOIS EDS with acceptable effort. The uses of other protocols such as CCSDS Space Packet Protocol or ECSS Packet Utilization Standard (PUS) are assets.

5.4. USABILITY

Interoperability is defined in ISO/IEC 25010 as the ability of a software system to be used by knowledgeable users to effectively and satisfactorily achieve the specified project objectives in a given operational context. We are going to extend this definition so that it is more appropriate for frameworks.

This attribute must connote a notion of conviviality designating the tools that the framework makes available to the developer to help him in his development and that it is capable of automatically executing simple, repetitive tasks or those subject to human error. We consider that this type of support tool has a beneficial, decisive and lasting influence on the success of a project because it also underlines a standardized process for building the software system.

This attribute is evaluated on the basis of the following sub-characteristics.

5.4.1. Appropriateness recognizability

Appropriateness recognizability is the ability of a software system to allow its users to recognize whether it is suitable for their needs.

In the context of a spaceflight software framework, this can be expressed by the presence of a table listing all the projects and programs implementing the framework. In this list would appear different information such as:

- Project/Program;
- Software class (A to F);
- Prime contractor;
- Cost estimation;
- Operating system;
- Hardware platform;
- Launch Date or TRL in case of TRL < 9.

We consider this type of table to be very important because it reassures stakeholders and encourages them to use the framework by allowing them to compare their project to a project already developed with the framework and to find similarities.

5.4.2. Learnability

Learnability is the ability of a software system to be easily learned and understood by its specific users to achieve specific learning objectives of using the system effectively, efficiently, safely, and satisfactorily in a specific context of use.

This attribute is important because it is closely related to ease of use; the easier a software system is to learn, the less training and time it will take for a person to use it.

In order to overcome the learning curve by training its employees, a software framework that incorporates proprietary or unfamiliar technologies will necessitate significant budget allocation from stakeholders.

Remember that stakeholders can include everything from government agencies to system integrators to small and medium-sized businesses with varying research budgets.

A SME, for example, may be able to devote one of its employees to studying a framework for a hundred hours, and the employee must be sufficiently confident in mastering the framework with the various technologies involved at the end of the study.

In the case of a framework, this can be assessed by the need to master more or less popular tools, libraries or computer languages to implement the framework.

5.4.3. Popularity

Popularity is a public favor, a reputation earned as a result of a well-known and popular software system. This feature is especially beneficial for open source frameworks that have a global community. You can assess the framework's popularity by using websites such as discussion forums, Google Trends, or the framework's hosting site.

A community that develops around a framework will attract a large number of users. When a framework's community is large and active, it has an advantage in terms of increased mastery of the framework. Because the community can offer its expertise and feedback, the impact of problems that a stakeholder may encounter when implementing the framework can be greatly reduced.

5.4.4. Operability

Operability is the ability of a software system to be working and monitored in a safe and reliable state according to specific requirements[54].
This may include that the software has been designed to function nominally without requiring a system reboot.

5.4.5. Human-Computer Interaction / User interface Aesthetics

User interface aesthetics is the ability of a software system to provide the user with an intuitive and pleasing graphical tool to achieve their goals.

In the context of spaceflight software operability, it is important to have a graphical tool to visualize the time tracking curves of certain received measurements, whether they are numerical values such as voltages or images. It should also be possible to run validation scenarios with sent commands and expected measurements in order to verify the operation of the system.
We will also consider graphical tools that can work in a remote development environment as an advantage [55].

5.4.6. Accessibility / Manage Complexity

Accessibility is the ability of a software system to be easily used by people with a wide range of characteristics and skills to achieve a specific goal in the context of a project.

This attribute is a valuable means of discovering issues that may arise for people with different needs.
It may be necessary for systems engineers or quality engineers to manipulate the framework to analyze or inspect certain aspects that are outside the domain of expertise of flight software engineers.

The number of programming languages that can be used to implement the framework influences its complexity. The system will become more complex as the number of languages increases. Furthermore, if the programming languages used have a small user base, the framework will be perceived as complex because there will be less support on online forums and fewer potential collaborators on the job market.

5.5. RELIABILITY

Reliability is defined in the ISO/IEC 25010 standard as the ability of a software system to perform specific functions in specific operational contexts over a given period.
This attribute is evaluated on the basis of the following sub-characteristics.

5.5.1. Maturity

Maturity is defined in the ISO/IEC 25010 standard as the ability of a software system to meet reliability requirements in nominal operation.

In the context of space flight software, we can evaluate the level of maturity in terms of the Technical Readiness Level (TRL) associated with it or with the different projects developed from the framework. As a reminder, Technology Readiness Levels (TRLs) are a type of metric used by NASA to assess the maturity level of a particular technology [56].

5.5.2. Availability

Availability is the ability of a software system to be operational and accessible when its use is required.
A software system with first-rate availability indicates that it will repair any malfunction so that the downtime does not exceed a specific time value. Availability is expressed as the ratio of available system time to total work time.

To determine system availability, we were tempted to talk about a service level agreement (SLA). However, this quality of service is determined not only by software engineering, but also by the entire system, which necessitates expertise in hardware engineering, radiation, mechanics, and other disciplines.
Therefore, we consider this service to be slightly outside the scope of what a framework can provide for the purposes of this study.

5.5.3. Fault tolerance

Fault tolerance is the ability of a software system to operate as expected despite the presence of hardware or software failures [57].

5.5.4. Recoverability

Recoverability refers to a product's or system's ability to recover directly affected data and restore the desired state of the system in the event of an interruption or failure.

Recovery can be classified into three types: functional by component redundancy, functional by alternative path, and degraded functional. Normally, these issues are the responsibility of the user's software system design, but some frameworks can make this difficult. If the framework's base architecture does not support these types of recoveries, the user must be able to integrate their own recovery logic.

In the first category, the framework shall allow the user to manually or automatically activate a redundancy unit in the event of a reported component failure in order to achieve autonomous operation through automatic predictive recovery.

Nonetheless, a large memory footprint may be required due to the formalism of a component specification. Furthermore, because component redundancy may not be possible in a context with limited memory size, the user must be able to configure a different method of recovering lost functionality.

If the first two types of recovery fail, system functionality must be reduced while system software control is maintained. The communication protocols used to send commands and receive telemetry messages must remain operational in order to reload the software or restore the system state.

5.6. SECURITY

Security is defined by the ISO/IEC 25010 standard as the ability of a software system to protect information and data so that people or other software systems have the degree of access corresponding to their type and level of authorization. The idea is that we want to control access to systems to which we can receive commands or send actions.
It denotes a capacity of the software system to stop and block malicious or unauthorized actions that could destroy the system as a whole.

Security includes authorization and authentication techniques, protection against network attacks, data encryption. Depending on the type of equipment, systems, satellite, payloads or instruments, on which the software is embedded, more or less formal requirements in terms of security may appear, but these are rather at the spacecraft level.
Indeed, the instrument control software can be directly integrated into the satellite on-board software or for reasons of real-time communication or bandwidth with instruments, the latter will tend to reduce the size of the data and their processing (serialization, encryption ...) making the security mechanisms almost non-existent.

So, the importance of quality attributes varies depending on the type of equipment. In the case of an instrument these will be considered but will not be of high criticality. This attribute is evaluated on the basis of the following sub-characteristics.

1. **Confidentiality** is defined by ISO/IEC 25010 as the ability of a software system to make its data accessible only to software systems authorized to access it.
2. **Integrity** is defined by ISO/IEC 25010 as the ability of a software system to prevent unauthorized access or modification of software applications or data. This attribute relates to errors that arise during the writing, reading, storing, transmitting or processing of data and that result in unintended alteration of the original data.
3. **Non-repudiation** is defined by ISO/IEC 25010 as the ability of a software system to provide evidence that actions or events have occurred, such that these events or actions cannot be repudiated at a later time.
4. **Accountability** is the ability of a software system to be accountable, to be held responsible or to meet specific requirements.

As we can see, it is difficult to evaluate the aforementioned security quality attributes in the context of a framework because they also depend on the hardware platform, operating system, and other ancillary software that the framework abstracts from and leaves to the user to select.

Not to mention that judging them correctly requires a fairly advanced level of computer security expertise, these attributes do not cover all the cybersecurity aspects associated with an open source framework. Indeed, because it is open source, a framework requires a cyber-secure development environment to ensure that malicious agents do not introduce vulnerabilities into users' software systems that could be exploited later.

We prefer to use some Open Source Security Foundation's cybersecurity assessment criteria [58], which are evaluated in their open source tool "Security Scorecards" [59]. This automated security tool detects risky practices in open source project development environments. It can be configured using GitHub actions and is available on the GitHub marketplace. It evaluates a number of software security heuristics and assigns a score from 0 to 10 to each checkpoint [60]. In this article, these checkpoints will be used to evaluate the security of a framework. The criteria are listed below.

5.6.1. Binary-Artifacts

The inclusion of generated executables in the source repository increases the risk to the user, who will frequently use the executables directly if they are included in the source repository, resulting in various risky behaviors. The generated executable (binary) artifacts pose several problems. They are not reviewable, which allows for the creation of outdated or malicious executables. Peer reviews typically focus on the source code rather than the executables, as it is difficult to ensure that the executables match the source code. Included executables may no longer match the source code over time.

Source code generated by other tools, such as Simulink, is tolerated in this context because it is less risky because it is easier to examine than executable artifacts. For an open source framework, it is best to remove all generated executable artifacts from the repository and allow the user to rebuild the project from the source files.

5.6.2. Branch-Protection

Framework administrators must enable the branch protection settings of their source hosting provider to avoid forced pushes or the suppression of important branches (GitHub, GitLab, etc.). Branch protection allows for the establishment of rules that govern the flow of work for the branches. Before accepting a contribution in a major sector, these safeguards may require the following rules to be followed:

- Requiring a code revision by a reviewer to significantly reduce the risk of a malicious agent introducing malicious code.
- Requiring the completion of continuous integration tests (project build, linters, tests, etc.)
- Prevent the forcing of a push, which can result in the irreversible deletion or overwriting of commits to the remote repository and rewrite the public modification history. This is accomplished in git by prohibiting the use of the "git push --force" command.

5.6.3. CI-Tests

The framework must have a continuous integration pipeline in place to run tests before merging pull requests from contributors. This continuous integration pipeline causes scripts to run tests, allowing developers to detect errors early on and, as a result, reduce the number of vulnerabilities in the repository.

5.6.4. CII-Best-Practices

The framework must adhere to a set of security-focused best development practices for open source software. Obtaining the Open Source Security Foundation (OpenSSF) best practices badge [58] is the quickest and easiest way to determine whether a framework adheres to best practices and, as a result, is likely to produce higher quality secure software. These best practices include:

- Providing a brief description of what the software does
- Informing users on how to obtain, provide feedback (in the form of bug reports or enhancements), report vulnerabilities, and contribute to the software.
- Provide a functional build system capable of automatically rebuilding the software from the source code (if applicable).
- Have a general policy that when significant new functionality is added, tests will be added to an automated test suite - Have a unique version identifier for each version intended for use by users following the SemVer or CalVer version numbering.
- Have at least one lead developer who understands how to create secure software
- Apply at least one static and dynamic code analysis tool to any proposed major production release.

5.6.5. Code-Review

Peer review of code should be required in the framework's repository configuration for each request to merge a new contribution. These reviews can detect a variety of unintended problems, including vulnerabilities that can be fixed immediately before being merged, thus improving the quality of the code. Reviews can also detect or deter an attacker attempting to insert malicious code. If the framework does not have enough reviewers, administrators should try to recruit other maintainers, preferably from different organizations, who will be willing to review each other's work.

5.6.6. Contributors

Contributors to the framework should ideally come from a diverse range of organizations, to give potential users of the framework confidence.

5.6.7. Dangerous-Workflow

The framework should avoid dangerous workflow patterns that exploit the workflow trigger, allowing malicious pull request authors to gain write access to the repository or steal secrets from it.

5.6.8. Dependency-Update-Tool

The framework should use a dependency update tool because outdated dependencies expose the framework to known vulnerabilities and make it vulnerable to attack (such as dependabot or renovatebot). These tools automate the process of updating dependencies by looking for obsolete or insecure requirements and issuing an update request if they are found.

5.6.9. Fuzzing

A framework shall have its own fuzzing infrastructure capable of running the Fuzz test harness continuously. Fuzzing is used to complement the explicit testing of functional requirements (unit and integration tests) of the mission. Fuzz testing is considered an important tool in the software development process to protect assets in security related and safety critical applications. Fuzzing is an effective method to detect bugs before they are discovered in operation and exploited by malicious agents. Fuzz tests detect programming errors that have nothing to do with the project requirements, such as [61]:

- C/C++-specific bugs:
  - Use-after-free, buffer overflows
  - Uninitialized memory uses
  - Memory leaks
- Arithmetic bugs:
  - Divisions by zero, int/float overflows, invalid bitwise shifts.
- Pure and simple crashes:
  - NULL dereferences, undetected exceptions
- Concurrency bugs:
  - Data races, deadlocks
- Resource utilization problems:
  - Memory exhaustion, hanging or infinite loops, infinite recursion (stack overflows).
- Logic bugs:
  - Discrepancies between two implementations of the same protocol.
  - Round-trip consistency bugs
  - Assertion failures

Unit or integration test harnesses are rarely written to try to catch these types of bugs, which are primarily responsible for security flaws and reliability problems.

5.6.10. License

The framework must have a license that informs users how and when they can use the source code. Any type of security review or audit is hampered by the lack of a license, which poses a legal risk to potential users. This license must be placed in the top-level directory in a .txt, .html or .md file named LICENSE or COPYING.

A framework with external dependencies that increase the number of different licenses will be rated lower. Indeed, too much diversity of licenses makes the analysis of limitations more difficult for the user and may require the use of a specialized license analysis tool.

5.6.11. Maintained

The framework must be actively maintained (for example, one commit per week for the previous 90 days). A non-active project cannot be patched, its dependencies cannot be fixed, and it cannot be actively tested and used.

5.6.12. Pinned-Dependencies

If a framework has external dependencies, it must declare all of them in a package format file: "conanfile.txt" for C/C++, "requirements.txt" for Python, and "package.json" for Javascript. A compromised external dependency may jeopardize the framework.

5.6.13. SAST

The framework should employ static application security testing (SAST), also known as static code analysis [62]. These tools (CodeQL, LGTM, SonarCloud) can prevent known bug classes from being introduced inadvertently into the code base.

5.6.14. Security-Policy

The framework must include a security policy notice. A security policy tells users what a vulnerability is and how to report it securely so that information about a bug is not made public. This security policy should be stored in the project's root directory as a SECURITY.md file.

5.6.15. Signed-Releases

The framework must cryptographically sign version artifacts to ensure their provenance.

5.6.16. Token-Permissions

Permission definitions for each workflow in a framework's yaml file must be set to read-only at the top level, and the required write permissions must be declared at the runtime level. Malicious agents can infiltrate the project using a compromised token with write access.

5.6.17. Vulnerabilities

There should be no open, unpatched vulnerabilities in the framework. Patches for any vulnerable components should be made available to users as soon as possible, as an open vulnerability is easily exploited by attackers.

5.7. MAINTAINABILITY

Maintainability is defined by ISO/IEC 25010 as the ability of a software system to be effectively modified to improve, correct or adapt to changes in the environment (hardware platforms, operating systems) and requirements.

Maintainability is intrinsically linked to the fundamental structure of the software system, its architecture type.

Excessive dependencies between components have a very negative effect on maintainability and are similar to the notion of "spaghetti code" in programming.

To remedy these problems, concepts based on the separation of responsibilities by discretizing the software system into components are the best way to achieve a high level of maintainability. This attribute affects not only the development process of the software system but also the organization of the team around it.

We will consider as advantages architecture models such as service-oriented architecture or microservices architecture that make the separation of responsibilities a foundation of their structure.

This attribute is evaluated on the basis of the following sub-characteristics.

5.7.1. Modularity

Modularity is defined by ISO/IEC 25010 as the ability of a software system to be decomposed into small, well-defined building blocks, so that changing one element has minimal impact on the other elements. These elements must be replaceable without affecting the entire software system.

5.7.2. Reusability

Reusability is defined by the ISO/IEC 25010 standard as the ability of a software system to be used in other software systems without modification. This attribute saves development time and therefore reduces costs by using previously developed, tested and criticality-certified components.

5.7.3. Analyzability

Analyzability is defined by ISO/IEC 25010 as the ability of a software system to effectively assess the impact of a change on one or more of its components, or to diagnose a system for deficiencies or causes of failure, or to identify components for modification.

5.7.4. Modifiability

The ISO/IEC 25010 standard defines Modifiability as the ability of a software system to be modified efficiently without introducing regression and degrading the quality of the existing product.

5.7.5. Testability / Verifiability

The ISO/IEC 25010 standard defines Testability as the ability of a software system to be effectively tested to ensure that it meets predefined acceptance criteria. This characteristic is essential for avoiding errors in complex and critical systems. It relates to the concept of operational safety by enhancing a system's maintainability, reliability, or availability.

Unit test development and maintenance are critical to the success of component reuse and must be supported in a cost-effective manner.

As a result, the development team must be liberated from the repetitive implementation of common unit test features such as assertions or mocks.

It is common practice to use unit test libraries, also known as unit test frameworks, to lighten and simplify the unit testing phase, whether they are open source or proprietary.

In general, these unit test libraries follow an assertion-based testing philosophy and provide several benefits, including:

- Make it mandatory for the developer to write explicit verification statements that assert whether a condition is true or false.
- Standardize the process of developing unit tests.

Therefore, we will consider that a framework that incorporates a unit test library into its development process has a higher degree of testability than one that does not. Similarly, a framework that includes a test suite guaranteeing 100% coverage in modified condition / decision coverage (MC/DC) will have a high testability score.

5.7.6. Supportability / Serviceability

Supportability is the ability of a software system to provide support tools that help identify and resolve problems. These support tools include, for example, diagnostic, analysis, monitoring, debugging or logging capabilities.

In the context of a framework, we will consider that any tool that automates repetitive tasks subject to human error increases its degree of supportability. For example, a tool could generate the toponymy of a component with skeleton codes in the case of a component-based architecture with well-defined semantics.

5.7.7. Scalability

Scalability is the ability of a software system to be extended without changing its existing code, thereby increasing its capacity and functionality as needed.

Scalable software remains stable while adapting to changes, updates, revisions, and resource reductions.

Depending on the type of architecture, there may be a defined scaling factor so that we know how the performance of the software system will change as we add functional units, and also what configurations might change for a minimum and maximum system size.

5.8. PORTABILITY

Portability is defined by ISO/IEC 25010 as the ability of a software system to be effectively transferred from one hardware, software or other operational or usage environment to another.

The software system's portability should be as seamless as possible and should not pose any behavioral problems.

Main factors affecting the portability of a software system are the dependency on the hardware platform or the operating system or the implementation language.

A software system with a high level of portability must be composed of several abstraction layers designed to limit the impact of hardware or software (operating system) platform changes [63].

It is due to the presence of this kind of layers that the porting of a software system is simpler and has a much lower cost than the development of a new implementation to deploy it on a new platform.

With regard to the degree of portability of a software framework related to the nature of the implementation language, there are several approaches.

Either it uses a programming language that is well known in the field, such as C. The disadvantage of this solution is that it may close doors if other, more powerful languages are developed in the future.

Or it integrates a device that allows to describe in an abstract way the discrete elements composing the system, in an implementation-agnostic way.

Thus, when designing the topology of the software system by discretization into components, the reasoning would be in terms of service data units, i.e. the nature of the data exchanged, rather than considering the problems related to the implementation in a particular programming language. The software system would then have a system for generating this abstract description of a component into a skeleton of code implemented in the specified language so that the developer can integrate the internal logic.

This attribute is evaluated on the basis of the following sub-characteristics.

5.8.1. Adaptability / Flexibility

Adaptability, or Flexibility, is defined by ISO/IEC 25010 as the ability of a software system to be effectively adapted to different or changing hardware, software or other operational or usage environments.

A software system needs to consider this attribute given the dynamic nature of mission requirements that dictate the environment in which the system operates [64].

5.8.2. Installability

Installability is the ability of a software system to be successfully installed and/or uninstalled in a specific usage environment. A framework's installability is influenced by a variety of factors.

- The framework's dependencies with the world outside the company's world must be listed.
- The framework's licenses should be as broad as possible.
- The framework must be deployable in a virtual development environment in which all dependencies are grouped together and no version conflicts with other third-party dependency versions exist.

5.8.3. Replaceability

Replaceability is the ability of a software system or one of its components to be replaced by another for the same use and the same operating environment.

The framework should not be overly coupled to the instantiated software system, allowing for relatively transparent changes to the framework. In theory, the development team would be responsible for keeping this coupling to a bare minimum, but the framework may require the use of framework-specific technology that cannot be modified.

6. SPACE FLIGHT SOFTWARE FRAMEWORK PARADIGM

In this chapter, we will characterize the spaceflight software framework paradigm, which is a set of principles and knowledge from software research, specifically spaceflight software. With this paradigm, we will be able to conduct the study in a more efficient, rational, and insightful manner.

6.1. ANALOGY WITH A CRAFTSMAN'S WORKSHOP

Before we begin, an analogy with a craftsman's workshop, and more precisely a blacksmith, will be used to demonstrate what a framework is and the various operational contexts it must deal with. Blacksmith was probably chosen unconsciously because the surname of one of the authors of the study refers to this profession (the name Farges comes from the French "forgeron" which means blacksmith in English).
The purpose of this description of a blacksmith's workshop is not to be accurate or exhaustive; rather, it is to highlight the similarities it has with a software framework. We sincerely apologize in advance to any blacksmith who may have found on this report and analogy and been offended by it.

A blacksmith is a skilled craftsperson who creates objects out of various alloys. The blacksmith's skills enable him to create a wide range of everyday objects from any era. Customers might be interested in cutting tools or decorative ironwork, for example. For the sharp objects, composed of a blade and a handle, they take infinity of forms like knives, swords, scythes, axes, and even katanas. While for the decorative ironwork objects, we could have gates, barriers or posts.

Depending on the market, the blacksmith may be required to perform more or less sophisticated work on a one-time or recurring basis. Consequently, the blacksmith's workshop must be well-equipped in order to produce high-quality work quickly and in accordance with the customer's needs.

This workshop will include various elements that correspond to the stages of the entire manufacturing process.

During the study phase, the blacksmith represents the final aspect of the object and creates a list of the materials he will require. He will need sheets of paper, pencils, erasers, and a computer with modeling software. But he also needs a variety of materials with properties that allow them to meet the end user's specific requirements.

During the manufacturing phase, hammers, anvils, hydraulic presses, clamps, and sanding or sharpening machines will be required. For the product's delivery phase, cartons, protective paper, and a delivery truck will be required.

All of these elements are linked to one another by a logical scheme that encompasses the entire manufacturing process of the work, from understanding the customer's needs to delivering the work to the customer.
The quality of each component of the workshop has an impact on the quality, time to market, and thus customer satisfaction.

Like a software framework for a developer, a blacksmith's workshop provides a workspace that allows him to do the work required more efficiently.

6.2. CONCEPT

We will define the concept of a software framework for the space domain because this research focuses on space flight software frameworks. This definition is intended to be global, progressive, and forward-thinking in order to determine the distinguishing features of what is known as a framework and what this concept should strive for in an iterative and incremental process.

A software framework is a digital platform that provides a centralized workspace for software engineering stakeholders. Within the same organization, these stakeholders come from a variety of professions, including systems engineers, software developers, and operators. The open source dimension adds the characteristic of having a common framework between different professions in different organizations. So here is an initial definition of a software framework.

"An open source software framework serves as a common workspace for the different actors and organizations involved in software development. It allows them to focus solely on the mission, the added value, of their product (science, performance, etc.) rather than redeveloping common software features."

The framework workflow defines a standardized process for developing and deploying applications in order to create a versatile and reusable software environment that provides specific functionality as part of a larger platform managed as a product line to aid in the development of software systems. The framework's infrastructure elements are integrated into a predefined workflow; in other words, the framework specifies not only the elements individually, but also their interrelatedness.

A framework allows for more granular reuse because it is organized as a set of features. The platform may include functionality that is superfluous for some missions due to its generic design; nevertheless, the users can alter the platform by tailoring it to tailoring to accommodate the unique aspects of their mission and obtain a mission-specific software system. A framework's reuse unit is thus not a single item, but rather a set of cooperating parts that combine to make the framework's important functionality easier to implement, commercially viable.

The framework's ability to be tailored is what makes it reusable as it can be used in a variety of operational contexts. In practice, different operational contexts of a project imply different operational requirements. The adaptability of a framework is critical to its reuse because it necessitates that its components be adaptable to different requirements. Components have adaptive capabilities in this sense, allowing them to change their behavior to meet specific mission requirements. Hardware platforms, operating systems, communication protocols, and standards are all specific requirements.

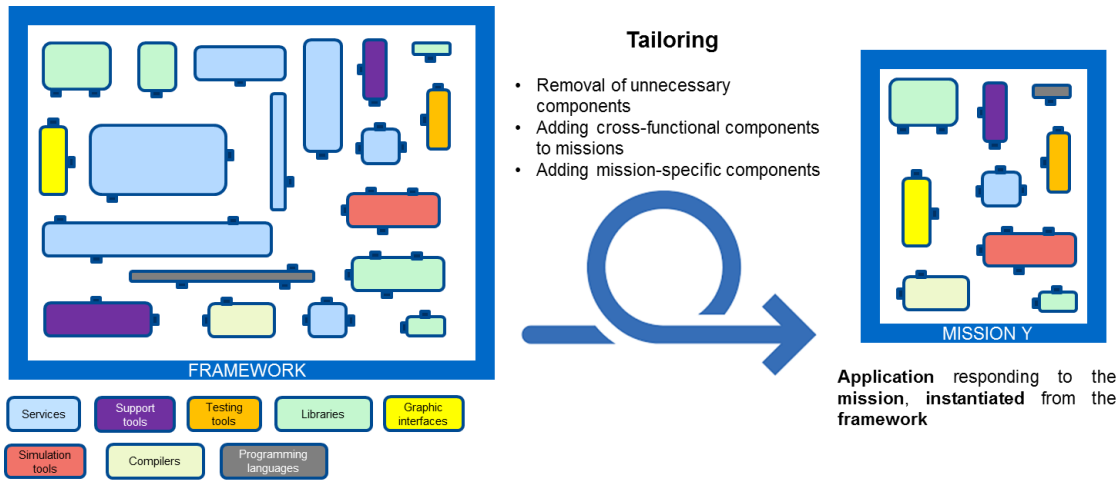


Figure 6-1: Overview of framework tailoring

6.2.1. Product Lifecycle Coverage

A software framework should aim to cover all phases of the life cycle of a software system, from feasibility analysis to in-flight maintenance, as well as development and validation.

6.2.1.1. Pre-study phase

Let us start with feasibility studies and project derisking phases. A recurring issue in embedded systems is the availability of non-COTS embedded platforms, which are only available very late and are out of sync with flight software development. This is especially true for advanced demonstrators and projects involving special-purpose instrumentation.
The disadvantage is that during the feasibility phase, which occurs before hardware design, the flight software team must be able to create mock-ups of the product.

These mock-ups enable to size the hardware platform by evaluating, among other things, the CPU load, the amount of non-volatile and volatile memory required, and the upstream and downstream data rates. This emphasizes the value of teams having a software testbed (like the GERICOS platform of LESIA [65], which provides an environment for rapidly designing prototypes by artificially reproducing the more or less real functioning of various hardware platforms, also known as simulation and emulation. The goal is to create digital twins of real-world systems using technologies such as QEMU [66] or Renode [67]. The digital twin concept is becoming more affordable with the rise of the IoT field, which provides open source tools for this purpose.

One interesting aspect is whether the framework supports artificial reproduction and whether it can be run in a virtual environment, whether remote or local, also known as a development container or remote development environment.

It is not difficult to imagine teams deploying a containerized image of the project on a development server from which they can work remotely from their laptops if the framework supports it. These virtual environments are comparable to GitPod [68], GitHub CodeSpace [69], and DevContainer in Visual Studio Code [70]. The team would be directly operational because all installation phases prone to human error or portability/compatibility would be eliminated.

Once the project container is operational, the team will be able to study system performance using flight software running on a simulated/emulated platform. The ready-to-use environment would provide a data control and visualization screen, displaying the curves of the characteristic measurements that could be easily exploited for benchmark, performance, and trade-off evaluation.
Another option is to switch from the simulated platform to the hardware platform or from the simulated instruments to the hardware instruments.

6.2.1.2. Development Phase

During the development phase, it is essential to be able to run a software system on a development machine, just as it is during the feasibility phase.
The team's first priority should be to discretize the system into components, ensure the viability of the established topology, and test mechanisms for changing operational modes, data processing, and other product features. Once the architecture is as mature as possible, the team will address common embedded system issues such as cross-compilation, memory optimization, remote communication, and remote debugging.

The framework should boost development productivity by supplying tools for automating repetitive, human-error-prone tasks such as creating a component/application code skeleton and corresponding unit test skeleton. A framework could include a tool that uses an MBSE approach to model the topology of a software system, such as the AADL modeling language.

This MBSE approach could make a system implementation in C, C++ or Rust transparent by using automatically generated skeleton code. This would provide a common environment for the system team and the flight software team to collaborate on a shared system vision. It appears to be easier, however, to fall into a strong coupling of the software system with a framework that has this functionality. If the tool, for example, provides complete logic auto generation for a component, developers may be tempted to use it to create all or part of the software system.
As a result, the software system is completely dependent on the source code auto generator, and the development team loses its capacity to manually optimize or instrument the code. This over-coupling must be avoided at all costs and kept to a bare minimum, because the framework's use may be discontinued if it has become too complex, is no longer maintained, or has a more powerful competitor. And if there is a strong correlation between the framework and the software system that has been instantiated from it, migrating to another framework will be extremely difficult.

Rather than frameworks, the software system should be correlated on programming languages that have proven themselves in their application domain. Indeed, programming languages are far more important, effective, and long-lasting communication vectors than frameworks for expressing programmers' thoughts. A programming language has a better chance than a framework of becoming the common denominator among all developers.

For space flight systems, a framework must allow the software system to be implemented in C or C++ or Rust languages could be considered, but the user base collaborating on internet forums does not appear to be large enough at the moment.

The framework should then provide a ready-to-use suite of pre-built and pre-tested applications that provide specific and shared services to a variety of projects, which the developer may or may not integrate into his system. It is critical to provide the source code of the applications as well as the unit tests that the user will be able to run in order to certify these applications on the hardware platform. Indeed, proving the absence of dead code with 100 percent code coverage is required on some high-criticality missions.

For decades, certain product development methodologies have been proven to reduce product time to market. Agile, DevOps and TDD (Test-Driven Development) are some of the terms used to describe these methodologies [71]. Their common goal is to create an iterative and incremental software system with recurring test campaigns. Consequently, unit and macro verification tools must be included in the framework.

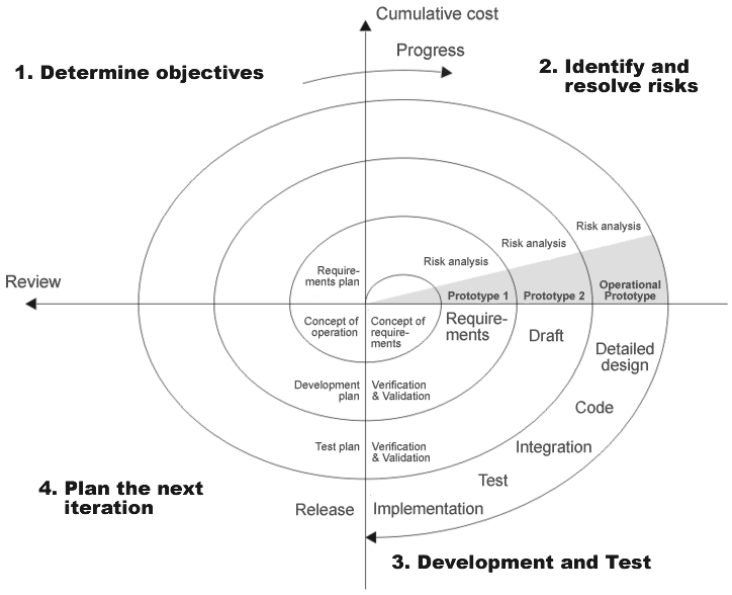


Figure 6-2: Schematic of Standard Spiral Model - © NASA

For unit tests, it should include a unit test sub-framework, which could be a third-party library or a framework-specific test library. For macroscopic requirements verification tests, it must provide a tool for running validation/regression scenarios with the definition of control and telemetry messages on a time scale and generating a results report. This package provides a certification suite for the software system under test.

6.2.1.3. AIT (Assembly, Integration and Test) Phase

At the risk of over-engineering the framework, we believe it could play a role in the AIT phase of the final delivered product. To summarize, AIT is a set of activities carried out by operators to ensure that the delivered product satisfies all mission requirements, not just those pertaining to software engineering. Traditionally, these activities are carried out on testbeds, which are built either in-house or through external contracting by a separate team from the team in charge of flight software systems.

The above-mentioned framework features, which fulfill the needs of both developers and operators, are positioned at the level of graphical interfaces. Some of the requirements that operators may have are listed below.

The operator needs to have a screen for real-time monitoring and control of the various test equipment's activities, including the onboard flight software. To describe the measurements received, various types of data visualization [72], such as mekko graphs, stacked bar graphs, column graphs, line graphs, scatter plots, image and video streams, and even 3D graphs, should be readily accessible.

The operator needs to have functionality for configuring tests with input/output based on a time scale, generating a test report, and automatically launching tests.

The operator's user interface should be a web application that includes client and server components [73]. The operator interacts with the client side, which is a user-friendly graphical interface. It is accessed via the user's web browser and does not necessitate any special operating system or device settings on his computer. This interface must be responsive and usable on both a smartphone and a laptop computer. Multiple operators, regardless of geographical location, must be able to track the progress of the tests at the same time. The graphical interface layout might be configurable and saveable based on configurations and users, which can be accomplished, for example, with Apache CoucheDB technology.

On the server side, the bare minimum is a main control center and a database. The control center serves as a gateway, converting the format of up and down messages exchanged with the equipment, as defined in a proprietary ICD or EDS, into a format that web technologies can easily exploit (such as BSON format). Databases can keep track of messages sent and received (particularly important for error messages), parameter files, and flight software binaries to be uploaded in the product.

6.2.2. Summary

To summarize, each software framework, like a blacksmith's workshop, has its own elements of varying quality that cover all or part of a space flight software system's life cycle. These elements are designed to help developers deliver applications on time, on budget, and with high quality. They can be made up of the following items:

- support programs, such as command line tools for auto-generating source code templates;
- programming languages;
- native or embedded platform compilers;
- ready-to-use suite of pre-built and pre-tested applications;
- pre-compiled libraries;
- simulation/emulation tools;
- unit test tools;
- validation/verification tools;
- graphical user interfaces.

The frameworks differ from each other in terms of functionality, usability and performance. Each framework has its advantages and disadvantages.

6.3. OPEN SOURCE MOTIVATIONS

Free and Open Source Software (FOSS) is an extension of the Marxist ideology of socialization of goods to the digital world, distinguished by the abolition of proprietary software and the adoption of community software. In this sense, it is a societal issue of knowledge sharing, technological independence and preservation of fundamental freedoms in the digital age.

6.3.1. Model

The publishers of this software adhere to an intellectual property model based on free licenses that allow them free and protected access to the source code of their software. By allowing users to study, use, modify, appropriate and redistribute the source code of these programs, these licenses demonstrate a certain liberalism. The software becomes a free-market resource that users can consume in unlimited quantities and whose consumption no one can restrict. They do, however, legally prohibit users from appropriating the efforts of the community and failing to provide the improvements and patches that they would have been required to implement. Despite this legal basis for the use of this software, it remains vulnerable to unscrupulous users who adopt a risky free rider posture.

This is a common failure of open market systems, in which some users do not contribute to a shared resource. This omission is extremely detrimental to the long-term development of free software. Licenses constitute a legal and moral contract of adhesion to the software between the user and its publisher. This model promotes altruistic moral values where everyone can contribute to the collective effort. The clauses of the contract establish a feeling of shared responsibility, a principle of contribution to the collective effort, which ensures a dynamic of perpetual development through the maintenance of the source code. The user is required to commit to the development of the community and to contribute to the maintenance of the software by reinvesting part of his earnings. These contributions can take several forms, both financial and non-financial.

In terms of financial forms, open source software can have a crowdfunding collective or crowdfunding campaigns. The budget raised can fund the addition of features, funding of source code audits, bug hunting contests or critical features, and funding of targeted advertising campaigns to attract more potential contributors. Or stakeholders can support by donating a certain percentage of avoided licensing costs, for example.

Conversely, for non-financial forms, this can be done by redistributing improvements developed through user’s projects. Stakeholders can also encourage professional engagement among their employees, as they are often personally invested in these communities and would meet Google's "20% of time" rule. This Google rule encourages employees to spend 20% of their time on activities they feel are most beneficial to the company [74].

Recently, the problems with Faker.js [75] and Log4Shell [76] have reminded us that funding for open source software remains essential.

6.3.2. License

It is essential to consider the license associated with the software that the user intends to use. The publisher may wish to exercise certain rights, permissions, and controls over how its software is used. Therefore, he may impose more or less strict restrictions and requirements through licenses that will contaminate the user code. There are mainly two types of licenses [77] permissive and copyleft licenses.

6.3.2.1. Permissive

Permissive licenses, also called "Apache style" or "BSD style," are the least contaminating open source licenses because they offer the greatest freedom of use.

Publishers use these licenses to make their software available to as many users as possible for a variety of reasons, including ethical, moral or interoperability concerns. These licenses allow the user to create derivative software with terms that differ from those of the original underlying software license. The terms of these licenses require that the copyright notice and license be included in the derivative products. The author provides the software "as is" and cannot be held responsible. The most popular licenses in this category include: MIT, Apache 2.0 and BSD 3.

6.3.2.2. Copyleft

Copyleft licenses, on the other hand, are more restrictive in that they limit the user's use of the framework. They make it illegal to sublicense software derived from the original. They ensure that users have the same rights and permissions when using software derived from the original. These licenses can be chosen by authors who do not want their software to be used by users for personal use and who do not want their use to be published as open source. If the user uses a copyleft framework, they will eventually be forced to publish their software, putting their intellectual property at risk. Among the most popular licenses in this category are GPL-3/2, LGPL-3 and MPL-2.0.

6.3.3. Innovation driver

Furthermore, the free nature of free software facilitates reaching a larger user base and challenging proprietary publishers' dominant positions, which are frequently the source of technological dependencies. Due to the widespread adoption of open standards, open source software enables software systems to be modernized, more efficient, and transparent by ensuring interoperability and reusability across multiple missions.

For many years, government institutions have believed that open source is the best way to achieve digital independence and that it can assist them in transitioning to a more inclusive digital environment that will stimulate the economy, foster competition, and encourage SMEs [78].

In a circular dated April 27, 2021 [79], the French government expresses its willingness to strengthen the opening of public source codes and the use of free and open source software.

In its 2020-2023 strategy under the theme "Think Open" [78], the European Commission outlines a vision for encouraging and harnessing the transformative, innovative, and collaborative power of open source, as well as its principles and development practices. According to the report, open source enables the co-creation and delivery of public services that connect seamlessly across organizational silos.

At the end of 2021, a group of European space industry prime contractors, equipment manufacturers, and start-ups (Airbus Defence and Space, Thales Alenia Space, Arianegroup, Sodern, Hemeria, Thrustme, and others) formed the Space Earth Initiative [80] and proposed ten measures to strengthen the European space sector. The eighth proposal [81], "Encouraging disruptive innovations through collaboration between industrial players and start-ups," fully reflects Open Source's primary goals.

In this context, it makes more sense than ever to advocate for open source frameworks that stakeholders can use to develop their businesses while openly contributing to the collective sector effort.

6.4. BENEFITS

Standardization and open source availability of a Spaceflight software framework offers a number of potential benefits for the development, deployment and maintenance of a software system [32]. If we were to categorize these benefits, we would differentiate between benefits on quality increase and cost reduction.

6.4.1. Quality-related benefits

In terms of quality increase, we have:

- Increase the quality in a homogeneous way of all flight software systems that inherit the same framework.
- Increase communication between space sector stakeholders at the software level (spacecraft, payloads, instruments) by enabling their collaboration within the same platform (communication of developers via GitHub, host of some frameworks, for example).
- Increase interoperability between stakeholders by standardizing software system interfaces (CCSDS SOIS EDS, CCSDS Space Packet Protocol, ECSS PUS...).
- Establish best practices in programming and design by working with the international community.
- Select the best product for a given task from a range of compatible components (availability of a ready-to-use suite of specific components).

6.4.2. Cost-related benefits

In terms of cost reduction, we have:

- Reduce uncertainty in the time-to-market, schedule, and cost of high-quality flight software systems by integrating previously developed software components.
- Reduce the cost of training newcomers. A newcomer is more likely to face an international open source framework than a proprietary framework. Students can also learn about them through software engineering training programs.
- Reduce the cost of maintenance and evolution of the platform over the long term thanks to the community that can add features or solve bugs without any additional effort from the company.
- Reduce the recurring cost of development by standardizing the framework's interfaces, allowing it to be reused between missions and establishing a common platform for several missions/programs.
- Reduce the cost of developing a proprietary framework by reusing a pre-developed, pre-tested international platform. A stakeholder may not have the funds and expertise to create a framework that is versatile enough to meet all of its products.

6.5. RISKS

There are some potential pitfalls or reservations about using a third-party framework; some are minor but should be mentioned.

6.5.1. Potential dwindle in pure programming proficiency

Developers' proficiency in the underlying programming language of the framework may wane over time. Developers gain expertise in the use of the framework without necessarily having in-depth knowledge of the programming language itself. Developers' knowledge of the programming language may be limited to a few libraries that are independent of the framework and part of the business layer, for example.

In the professional world, this is more nuanced. This would be true if the framework had no bugs or if the developer did not have access to the source code and the framework was simply used as a black box. If, on the other hand, a bug appears in the user's software system, the onus is on the user to understand, isolate and resolve it. Whether the bug is caused by the framework, the user's use of it, or another component of the source code, the developer will require programming skills to fully understand how things work under the hood and fix it.

6.5.2. Limited adaptability

A framework's processes and implementation may be too inflexible, too rigid, to adapt to the specific usage needs of a mission. To address interoperability issues, a mission may be forced to use specific technology stacks, communication standards, or compressions. These technologies are unlikely to be supported by default by a framework. As such, if the framework's design does not allow for the change of this type of technology, the mission's developers will be forced to choose a framework better suited to their needs.

6.5.3. License

Particular attention must be paid to the licenses that accompany the frameworks, since a legal framework governs a stakeholder's use of OSS. If the license requirements are not met, the stakeholder violates the intellectual property of the copyright holder and is held legally responsible. Therefore, open source frameworks with licenses that are as permissive as possible, such as MIT, are to be preferred to copyleft licenses or frameworks with a slew of licenses that require a potentially complex risk assessment [82].

This is especially significant because some of these licenses may forbid the use of these frameworks in military applications. It is critical to remember that the defense sector is and will most likely remain a major customer for space applications and services such as earth observation (EO), satellite communication (MilSatCom), and so on. Even if a product is primarily intended for the private sector, "anti-defense" licenses limit the number of potential customers and may jeopardize the product supplier's business.

6.5.4. Tight coupling with the framework

Despite the fact that the framework's authors claim to be "community driven," as Uncle Bob would say, there is an asymmetrical relationship between the framework and the user code. The roadmap of the framework may gradually drift away from the user's concerns by failing to address their most pressing needs. Users are compelled to update their versions on a regular basis in order to receive the most up-to-date support and patches. The user must adhere to both his project's time constraints and the underlying framework's release schedule. This is especially true when the user's application is tightly coupled to the framework. Otherwise, the user will be free to evolve at his or her own pace, or even to switch to a more appropriate framework.

A very interesting argument in this regard is made in [5]. The authors favor library integration over frameworks precisely because of the differences in the associated coupling points. Too much coupling introduces inflexible interactions between the components of the user code and makes it more difficult to replace its architecture when it needs to be changed.

Libraries have more of a utilitarian role for the developer because they provide him with a suite of classes or functions that he is free to call according to his needs. Developers therefore seem to have relative peace of mind when it comes to integrating an update to libraries because they are not very sensitive coupling points.

The philosophy of frameworks is different; they are the ones that invoke the user code. Because a framework serves as the basis for the user code, the coupling points are stronger. Because of this strong coupling, developers are more nervous about incorporating updates to a framework.

All of this contributes to the process of updating tools in the user code. Framework updates are considered push updates, while library updates are pull updates. A tip from the authors on this topic is:

> "Update framework dependencies aggressively, update libraries passively."
> – "Building Evolutionary Architectures" [5]

7. OPEN SOURCE SOFTWARE FRAMEWORK ADOPTION STRATEGY

According to the book "Building Evolutionary Architecture" [73], many companies consider software engineering as non-critical, similar to a heating system, or as a support activity for other departments such as electronics. The software engineering department may have a minor role in the overall organization of these companies. Software engineering expertise may be dispersed across several departments, or there may be no position dedicated to software system architectures, leaving the developers to establish the software architectures for products.

When software engineers discuss software innovation with a company's technical authority or executives, the latter may perceive them as craftsmen trying to sell them their revolutionary but overpriced technology. Depending on the quality of their arguments or the attention they receive, these software engineers can be mistaken for people trying to defraud their interlocutor.

For these reasons we will attempt to provide elements that will help architects and developers sell the use of open source frameworks in their space ventures. These recommendations address technical, business, and organizational concerns. They can also help dispel the sort of aura that surrounds the field of software engineering for spaceflight.

7.1. HETEROGENEOUS SPACE INDUSTRY FABRIC

The space industry fabric is made up of different types of organizations that respond to innovation in different ways. These organizations include both start-ups and established companies. In his book of the same name, Clayton Christensen describes the phenomenon "The Innovator's Dilemma" [83]. He demonstrates how companies with well-established markets are more likely to fail, whereas more agile startups respond better to the evolution of an ecosystem.

On the one hand, we have established companies with a large historical customer base and high expectations for annual revenue. On the other hand, there are startups that don't have these concerns and can therefore focus on innovation and cutting-edge technologies.

There are two types of innovation: continuous innovation and disruptive innovation. Established companies have the prerogative of so-called continuous innovation. It consists of evolving an existing product to better meet the needs of their customers while remaining competitive in their market niche. Disruptive innovation, on the other hand, is the prerogative of startups because it implements breakthrough technologies. They seek to exploit new market niches rather than satisfy existing customers who may be resistant to disruptive change. Startups go so far as to impose these new technologies on all sectors by developing their activity on these new market niches.

If a company wants to deploy a disruptive technology, such as an open source framework, it should start with new products rather than old ones. Implementing this technology on an existing product is a continuous innovation and may not please previous customers. Therefore, it is better to use it on new products for new markets with new customers. Due to a variety of barriers, organizational adoption of new technologies is slow or non-existent. The presence of legacy systems from an organization’s past, as well as archaic engineering practices, are the most significant of these barriers.

Consequently, some organizations are more conservative than others, taking a "don't touch it until it works" approach. This mentality prevents them from staying ahead of the competition and being at the forefront of new technology trends. If these organizations continue to think this way, they will fall behind as the space market evolves. Digital transformation improves efficiency, productivity and agility by using new technologies to deliver more innovative and adaptable solutions.

> “Innovation distinguishes between a leader and a follower.”
> – Steve Jobs

Startups are not affected by legacy systems, but they can be affected by archaic engineering practices introduced by their talent pool that do not adhere to current best practices. Established organizations, on the other hand, have a different story to tell, as they are additionally burdened with legacy software systems. We will focus on the latter.

Adopting a new framework will be more difficult for players that have long provided software systems and have developed specific development processes, support, operating or automation tools. These stakeholders will have little or no incentive to change their current software systems for at least three reasons. These systems serve the right need, are based on organizational functional silos, or are tied to historical customers.

Replacing aging legacy systems with new systems can have serious implications for an organization. Old software systems are part of the company's history. Replacing them may require training on the new system or, in the case of software engineering, reorganizing the company to better reflect the new system. There is also the cost of maintaining the old systems and the risk that the new system will fail when the old one has already proven itself. In this context of technological legacy, it is difficult to argue for the cost savings of an open source framework over development processes and technologies that an organization has already mastered.

How can we accept the responsibility of creating a recurring product based on an open source framework over which we do not have full control and whose feasibility is questionable? This question is not tied to any particular framework, as other, more successful frameworks will undoubtedly emerge over time. The question arises, from a broader perspective, of the introduction of an external framework and the mindset to embrace it.

There is no question that replacing the old system in a single day is impossible. We need to mature and test the new system to be convinced of its viability and to show that it corrects the flaws of the old one. A specific engineering device is used to evaluate the feasibility of a technological innovation. This device is the technological demonstrator. It is comparable to the new products on which we should deploy the new technologies. They seem to us to be the most qualified candidates to convince a company to use an Open Source framework.

7.2. TECHNOLOGICAL DEMONSTRATORS AS FIRST TARGET FOR INSTANTIATION

These demonstrators are the result of an investment in disruptive innovation aimed at providing stakeholders with operational superiority based on technological superiority. They do not always meet an operational need, but they allow the application, development and validation of existing or future technologies. The frameworks are completely consistent with this dynamic and simplify the creation of demonstrators, allowing for greater efficiency in program and project preparation as well as better risk-mitigation of subsequent developments.

Before an organization can be considered mature in its use of an open source framework, the process of evolving the demonstrator must be incremental and include a list of tasks to be accomplished.

A ROM Costing [84] with a long planning cycle seems to be one of the best solutions to start this adoption work and get an idea of the required budget. In this setup we need to keep our options open and avoid making irreversible decisions based on the selection of a specific framework. This ROM Costing should be broken down into early deliverables to test the feasibility of architecture and development infrastructure choices. This allows us to avoid frameworks that require a large upfront investment, such as support contracts, training programs, etc. This is significant because the framework will not demonstrate that it attempts to address the existing problem until it has been thoroughly tested and validated by feedback. Hence our idea of deferring these more or less important investments to the appropriate time.

To initiate this costing activity, we can make the following assumption. The evolution process of a demonstrator logically starts with the development of its software system on a host machine to avoid the problems associated with embedded systems such as cross-compilation, communication and remote debugging. Then, the evolution continues on commercial hardware platforms (COTS) with evaluation kits, common communication protocols such as Ethernet. And finally, the evolution continues on resource-constrained space hardware platforms and space communication protocols such as Spacewire.

This activity presents a potential vision of the product to the executive by providing a first draft of its roadmap, with the necessary budgets for each step. Key decision points in this roadmap could include:

1. Create a proof of concept for an experimental application that will allow us to instantiate a framework (TRL 2);
2. Prototype this idea on a host machine by creating a basic first iteration of the idea (TRL 3);
3. Make prototyping more difficult by shifting to COTS hardware platforms with turnkey tools/environments for application deployment (TRL 4);
4. Increase the complexity by transitioning to hardware platforms capable of being launched into space (TRL 5);
5. Perform a demonstration of space flight (TRL 6).

All these steps cannot be taken at the same time. They invite us to break things down and reconsider technical choices by gradually increasing complexity. A DevSecOps environment must be present from the beginning of development because it allows continuous testing of the product as well as knowledge of its performance and limits, particularly with the help of a digital twin. This environment must also allow automatic certification of the product at each stage of maturity.

This roadmap encourages an experimental mindset with room for empiricism. It is akin to the MVP (Minimum Viable Product) methodology developed by Eric Ries in "The Lean Startup" [85] or to the MLP (Minimum Lovable Product) methodology. These approaches from the startup ecosystem are excellent learning resources.

By organizing a technical demonstration at each delivery, it is possible to show the evolution of the product to the different stakeholders. This is essential because stakeholders can be both internal (such as the executive) and external (such as investors or customers). These demonstrations allow us

to tease and maintain their interest while showcasing performance and formulating improvements for potential future applications of the envisioned product.

These approaches help the development team better understand the latent need of a niche market by integrating stakeholders into feedback loops and maximizing feedback. The development team is in direct contact with the end users, allowing them to understand the impact of their work.

> “I think it's very important to have a feedback loop, where you're constantly thinking about what you've done and how you could be doing it better.”
>
> – Elon Musk

Once matured and proven in a project, the framework will be able to demonstrate its benefits within the existing system. But what are the next steps to popularize this success across the organization?

7.3. TECHNOLOGY ADOPTION LIFE CYCLE

Open source frameworks, like any other technology, are confronted by the sociological model of the technology adoption life cycle [86]. When applied to our problem, this model describes the adoption of an open source framework in an organization using the demographic and psychological characteristics of adopters groups. This model is depicted by a bell curve with several population categories, which are described further below [87].

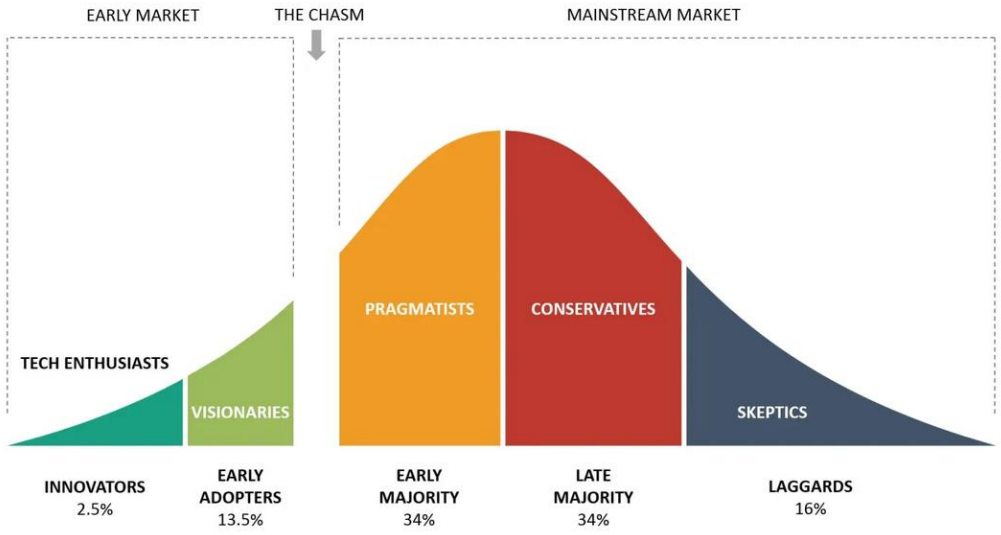


Figure 7-1: Technology Adoption Lifecycle © unknown

Innovators are the first group. They are the ones who are most interested in and knowledgeable about technology. They are the most daring and aggressive in their pursuit of new technology stacks because they are essential to their existence. They do not belong to the dominant socio-professional groups and represent only 2.5% of the population.

Then there are the Early Adopters, who use a new technology at an early stage in its life cycle. Unlike innovators, they are not technology enthusiasts. They are more integrated into social and professional systems and are more likely to be thought leaders. It is estimated that this group represents 13.5% of the population.

Next is the Early Majority, a more conservative population that is open to new ideas and eager to see how a new technology performs. This demographic group would make up 34% of the population.

The Late majority is generally older and has been part of the organization longer. They accept new technologies out of economic necessity or because of increasing social pressure. They are risk averse and will not adopt new technologies until the majority of their social system does. This demographic group would represent 34% of the population.

Finally, the Laggar are traditionalists who are the last to adopt new technologies. They are concerned about the past and want decisions to be made with the previous generations in mind. This group would represent 16% of the population.

> "Change is the law of life. And those who look only to the past or present are certain to miss the future."
>
> – John Fitzgerald Kennedy

These percentages can vary greatly depending on the profile of the company. Consider a defense company that is subject to stringent cybersecurity regulations and effectively isolates itself from the rest

of the world. Conservatives or skeptics may predominate in this type of company. One can even expect the terms "Open Source" or "Framework" to sound like buzzwords in these companies and put people off. An open source startup, on the other hand, can expect to have many more innovators or visionaries.

In any case, we can conclude that it is necessary to evangelize a wide range of people for an open source framework to become mainstream in an organization. Each population group is separated by greater or lesser chasms, as people prefer to listen to those within their own socio-professional networks.

The largest gap exists between the early adopters and the early majority. By getting the early majority to embrace the open source framework and make it their own, the organization will be able to close this gap and adopt the technology. To be convinced, the majority must have confidence in the new system and be confident that it works and makes their jobs easier.
For this reason, it is critical in the short term to implement an open source framework on new products and use their eventual success as a foundation for trust for other socio-professional groups. The new system has the potential to spread by capillary action to other projects in the program where the framework has been implemented in the medium term. Finally, it has the potential to spread to all of the company's products in the long run.

7.4. OPPORTUNITY FOR STEM-RELATED PUBLIC ENGAGEMENT

Before moving on to the implementation of some of these frameworks, we want to make a brief aside in the spirit of innovation and openness that we are trying to encourage.

We believe that these research activities would be an ideal opportunity for a STEM-related public engagement activity. This public engagement would be done through a collaboration between a private sector actor and a STEM educational institution. It could take the form of a company-integrated applied research lab, bringing together professionals and students.

This laboratory would serve as an innovation and training platform, allowing students to contribute to research and improvement projects proposed by the company. The emphasis would be on using software engineering principles to solve practical problems related to new technologies, such as open source flight software frameworks. The research results would benefit both the company and the international community through open source work, as well as students.

This laboratory could more easily benefit from public investment programs because of its open nature and the donation of all or part of the work to the public domain. This is a method of ensuring that public funds allocated to a private actor benefit society to the greatest extent possible.

8. SPACE FLIGHT SOFTWARE FRAMEWORKS

This section gathers the presentations, implementations and evaluation of each framework selected for this study. The versions of the frameworks on which the study was carried out are: 2.0.1 for F Prime; Caelum-rc2 for cFS and 9.1-beta2-64bit for TASTE.

8.1. NASA F´

8.1.1. Overview

F´ is a free, open-source, flight-proven framework developed at NASA's Jet Propulsion Laboratory (JPL) since 2013 and matured through several JPL projects between 2014 and 2017 [88]. JPL is a federally funded research and development center managed for NASA by Caltech (California Institute of Technology). It is suitable for small-scale flight software systems such as small satellite, payloads, and instruments. Anecdotally, F´ was used to develop the "Ingenuity" Mars Helicopter whose first flight took place on April 19, 2021 and is considered a great success by JPL [89].

F´ has a small but growing community, composed of universities (Georgia Institute of Technology for example), private companies and independent developers who communicate via the GitHub platform, a website where the framework is hosted and managed.

Since its release as open source, the project has been continuously maintained by a Caltech development team. The maintenance team, in charge of governance, accepts and encourages the open-source community to contribute. When a contribution is proposed, the maintenance team reviews it and incorporates it or not without the contributor having to sign a contributor license agreement.
In order to verify that contributions do not break the build chain, F´ has a continuous integration where it checks for a smooth build merged into an Ubuntu image instantiated with Docker.

F´ provides a fairly complete user documentation and also provides several tutorials available to understand the different concepts that are not necessarily referenced in the documentation.

Regarding the supported operating systems are for the moment Linux, VxWorks, MacOS, baremetal (without OS), and POSIX-compliant OSes. Regarding its portability on hardware platform, the framework ran on a wide range of them: Sphinx, Sabertooth, Snapdragon, PPC (RAD750), LEON3, x86, ARM, MSP430.

The framework includes several key features:

- Component-based architecture;
- C++ framework providing basic capabilities common to all embedded software;
- well-defined semantic to abstractly describe a component;
- Tools to generate the code skeleton of a component based on the abstract description;
- Collection of ready-to-use components;
- Command/telemetry GUI;
- Testing tools for unit and integration testing.

8.1.2. Ecosystem

8.1.2.1. Prerequisites

Since F´ includes two Python support tools, it is best to create an isolated Python virtual environment before cloning the project, although it is possible to do so later. Since these environments have their own installation directories and do not share libraries with other environments, multiple versions of a package can be installed without conflict on the user's computer.

After activating the environment, we can install the project and dependencies, including the fprime-util and fprime-gds python support tools:

- fprime-util enables developers to automate repetitive tasks and launch various configurations of the F Prime deployment build chains.
- fprime-gds provides a graphical user interface (GUI) and command-line interface (CLI) for communicating with F Prime deployments, allowing developers to send sequence of commands, and receive telemetry messages.

8.1.2.2. Design phase

The framework's development process requires the developer to take a "design first" approach, which forces the developer to break down their software system into components and interfaces between them. Through the use of an XML markup language specification, this design phase is intended to be as language agnostic as possible. The developer abstracts from any paradigm associated with a specific programming language during this design phase to facilitate the ontological approach to discretizing the software system, including its deployment topology, components, and interfaces.

While system design is approached top-down, specification implementation is approached bottom-up. First, the component interface data types must be defined, then the components themselves, and finally their integration into the deployment topology.

These specifications must be kept in separate files with the suffix "Ai.xml." Every specification, whether an interface specification, a component specification, or a deployment topology specification, must adhere to a strict formalism. In these formalisms, which are expressed using the RelexNG and Schematron schemas, the structure of an XML document can be specified using rules and assertions. These schemas impose a document's global structure as well as certain intrinsic constraints that the document must satisfy in order to be considered valid. We have a suffix for each type of specification:

- "PortAi.xml" for inter-component interfaces;
- "Serializable.xml" for serializable interfaces;
- "ComponentAi.xml" for components;
- "EnumAi.xml" for enumerations:
- "Array.xml" for arrays; and
- "TopologyAppAi.xml" for deployment topologies.

8.1.2.3. Implementation phase

After specifying a component and its interfaces in the XML files, the developer must use a code generator to generate the source files. The code generator generates two source files in the C++ implementation language for each object specified in an XML file. These source files, which have the suffixes Impl.cpp-template and Impl.hpp-template, contain the stubbed C++ implementation of the objects. Then the developer can implement the component logic in these files and remove the "-template" suffix to add the file to the build chain and also get the synthaxic coloring.

It is also possible to generate implementations of stubbed test objects, which can then be linked to the objects to be tested. These test objects provide a set of unit tests to validate the functionality of an

object and to obtain source code coverage. The GoogleTest library is used to implement these test components.

8.1.2.4. Deployment phase

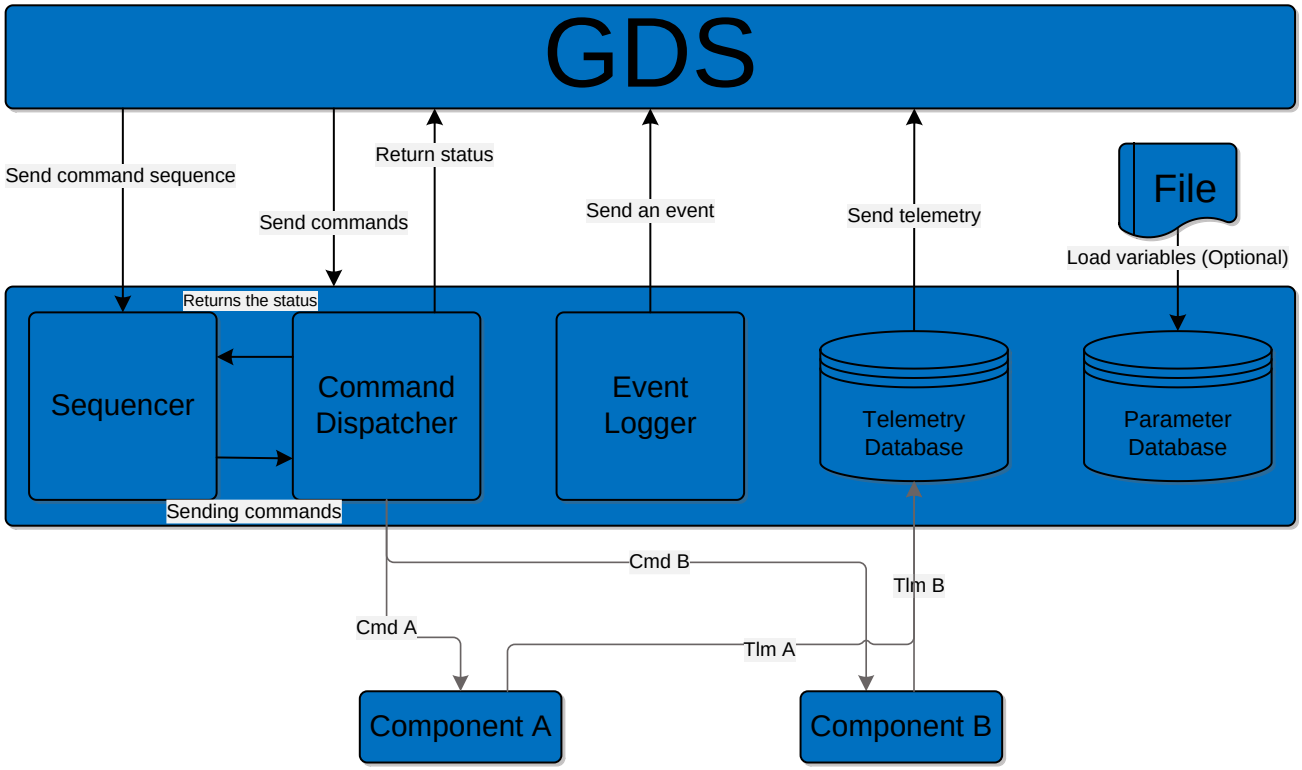
Once all the components are in place, the deployment topology must be specified as it describes the interconnections between each component and ensures that the software system functions as required by the mission.

First, the instances of the components must be defined in a Topology.cpp file. This file contains a function that allows us to do tasks like initialize the components, register commands, events and telemetry messages, and load parameters. It is also in this function that we will refer to as an auto generated function that performs component connection routing and is described in a file suffixed "TopologyAppAi.xml". We specify all of the topology's interfaces in this XML file of well-defined formalism. This file complicates the process because it is the most difficult to complete as it is edited manually, and human errors are common in total control.

At the time of finalizing this paper, the FPP (F Prime Prime) tool, which is included in the recently released version 3.0.0 of the project, greatly simplifies the specification of the topology and its components.

8.1.3. Architecture

F´ is based on a component-based architecture which relies on the following concepts: ports, components and topologies. These three items allow us to build a complete embedded system.



8.1.3.1. Port

A port is a type of service data unit that serves as an access point for a component. It allows typed interfaces to pass through, but only to other ports of the same type. Data of simple structure, simple number, as well as complex arrays or structure, can be transferred between components via ports.

When creating a port, the user must specify the communication direction, i.e. whether it should be in or out. In contrast to an input port, which can only connect to one other port, i.e. "1 to 1," an output port can connect to multiple input ports, i.e. "1 to n." A port can have several properties assigned to it:

- "Synchronous" refers to cyclic activation of a component's activity;
- "Asynchronous" refers to sporadic data reception to allow episodic triggering of a component; and
- "Guarded" to prevent concurrent execution of a component with other interfaces.

8.1.3.2. Component

The term "component" has the same semantics as it does in software architectures: it is a building block that executes a specific part of a software system. At the implementation level, a component is a C++ class that defines its data and methods, such as port management. A component can be one of three types.

- “Passive”: the component has no thread and cannot be triggered asynchronously. The component's execution must be done sequentially by another component, i.e. the calling component provides the execution context.
- “Queued”: the component lacks a thread but does have a queue. It handles asynchronous commands and port invocations, but the user must implement at least one synchronous port

invocation that offloads and handles queue messages. The invocator provides the execution context for this and any other synchronous invocation.

- "Active": the component has both an execution thread and a queue. The thread distributes queued port calls based on the execution context of the thread.

8.1.3.3. Topology

A software system's topology represents the design properties of its deployment, such as the components, their interconnections, and their relationship with the hardware platform. The topology encapsulates the software system, allowing data flows to be seen between its components regardless of their physical layout. A topology extract from a JPL presentation is depicted below.

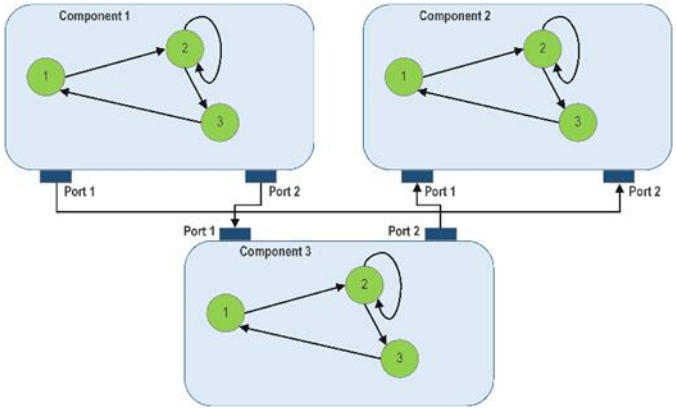


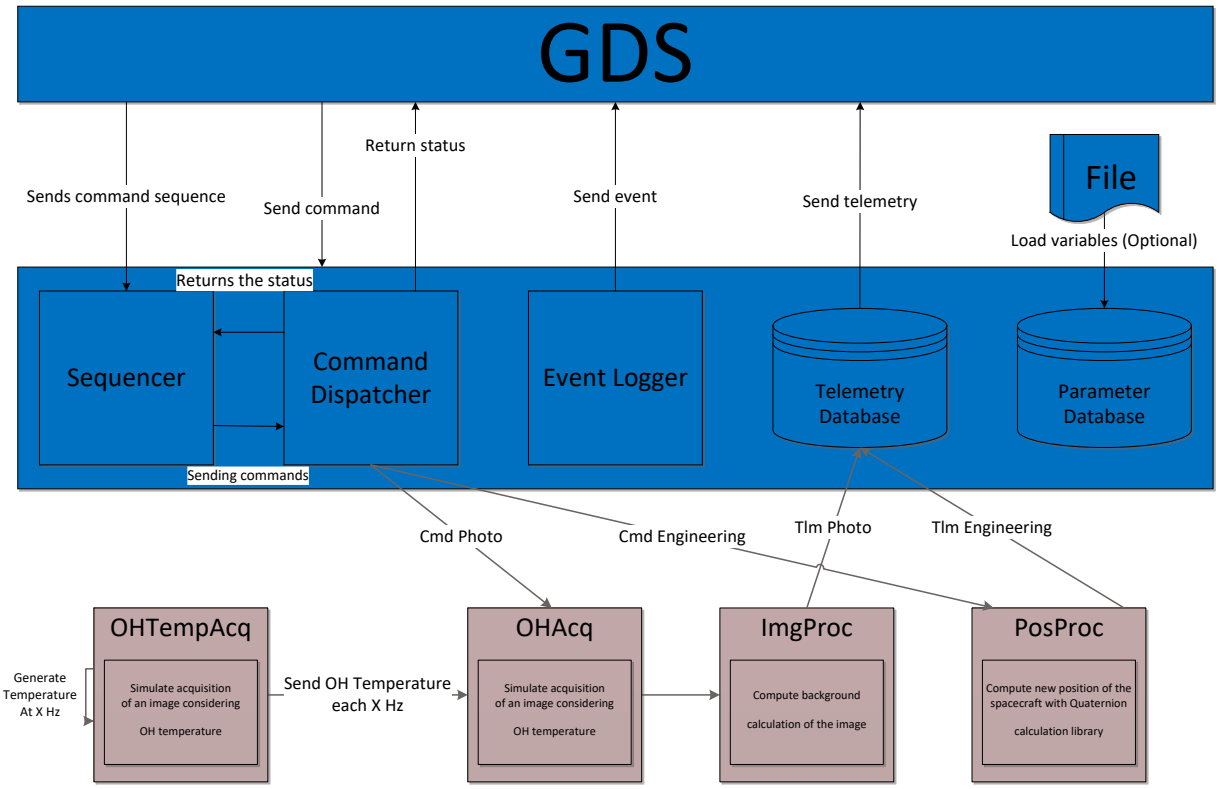
Figure 8-1: Example of a topological representation © NASA JPL

The topology's components are instantiated at compile time, and their interconnection becomes active at system initialization time, i.e. at the start of run time.

8.1.4. Proof-Of-Concept

8.1.4.1. Topology design

Here we can find a diagram representing the architecture of our proof-of-concept:



8.1.4.2. Create component

To avoid having to start from scratch, we used XML files of components previously specified in the examples to specify our components. We were unable to use the fprime-tools version that included the "fprime-util new" command, which uses CookieCutter templates to assist with component specification. This version arrived after we completed the proof-of-concept development.

To demonstrate the process, we will use the component in charge of processing a position "PosProc" as an example. To better understand the component's XML file, we decided to export each class of element in a separate file. So we have a file for each type, a file for each command, and a file for each telemetry channel.

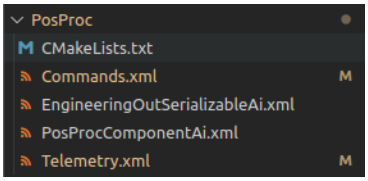


Figure 8-2: Component's taxonomy

By doing so, the component's global specification file, which serves as the entry point for the code auto-generation chain, becomes easier to understand.

```
MissionSpecificComponents > PosProc > PosProcComponentAi.xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="../../Autocoders/Python/schema/ISF/component_schema.rng" type="application/xml" schematypens="http://relaxng.org/ns/structure/1.0"?>
<component name="PosProc" kind="passive" namespace="PoC">
    <import_serializable_type>MissionSpecificComponents/PosProc/EngineeringOutSerializableAi.xml</import_serializable_type>
    <import_dictionary>MissionSpecificComponents/PosProc/Telemetry.xml</import_dictionary>
    <import_dictionary>MissionSpecificComponents/PosProc/Commands.xml</import_dictionary>
    <comment>Component TC Engineering</comment>
</component>

```

Figure 8-3: XML component's specification

After specifying the component's attributes, we must use the fprime-tools command to generate the stubby C++ implementations: "fprime-util impl." We get two template files, which we must change to implement the component logic, as intended.

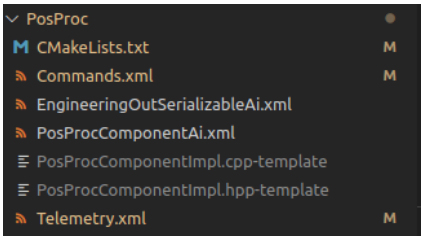


Figure 8-4: C++ implementation templates

The placeholder for the handler of the Engineering command is supplied below, which allows us to activate the component's execution.

```
45  // ----------------------------------------------------------------------
46  // Command handler implementations
47  // ----------------------------------------------------------------------
48
49  void PosProcComponentImpl ::
50    CmdEngineering_cmdHandler(
51        const FwOpcodeType opCode,
52        const U32 cmdSeq,
53        F32 x_start_pos,
54        F32 y_start_pos,
55        F32 z_start_pos,
56        F32 x_move_pos,
57        F32 y_move_pos,
58        F32 z_move_pos,
59        I32 rotation_angle,
60        I32 rotation_axis
61    )
62  {
63    // TODO
64    this->cmdResponse_out(opCode,cmdSeq,Fw::COMMAND_OK);
65  }
```

Figure 8-5: Engineering command handler

The placeholders include invocations to the component's output ports, allowing the developer to quickly create and compile their component. The created components can be directly inserted into the topology, and the developer can add additional functionality to each component iteratively.

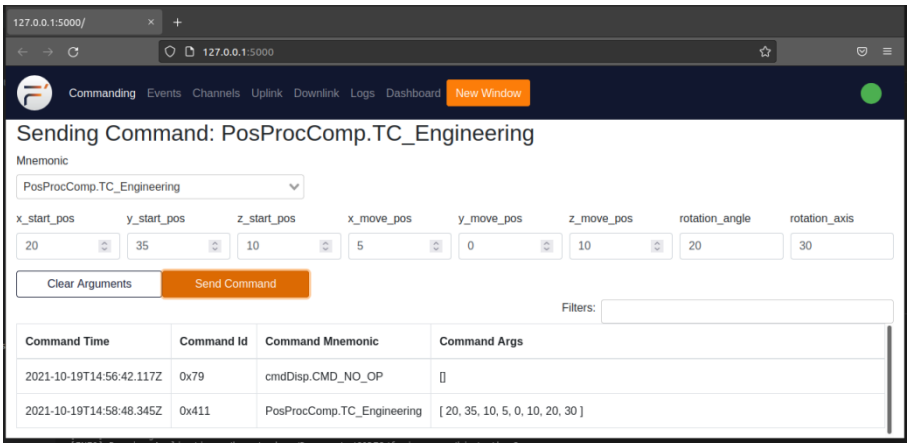
8.1.4.3. Adding the component to the system topology

Once each component has been implemented, either stubbed or with its full logic, it must be integrated into the software system's topology in order to be part of its deployment.

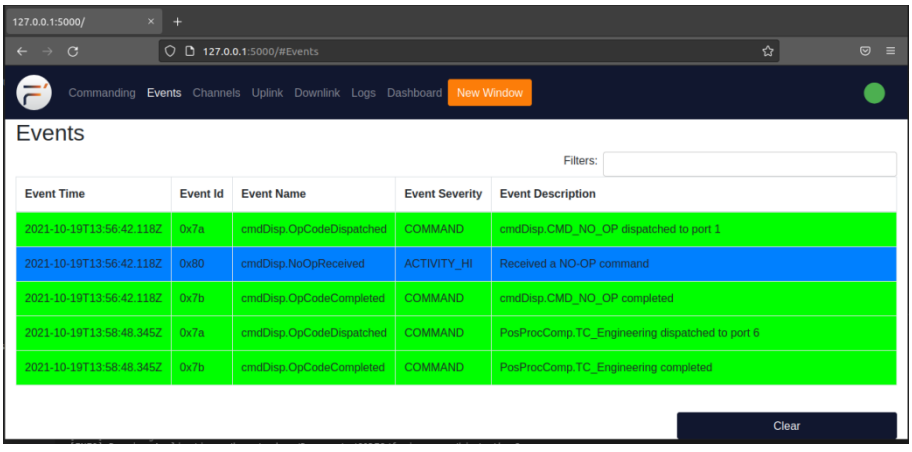
To accomplish this, each component must be defined in the Topology.cpp file and declared in the Components.hpp file. The component interconnections are currently specified manually in the RefTopologyAppAi.xml file. This XML file generates the RefTopologyAppAc.cpp file, which contains the function for defining component connections during software system startup.

8.1.4.4. Build the system and use the Ground System

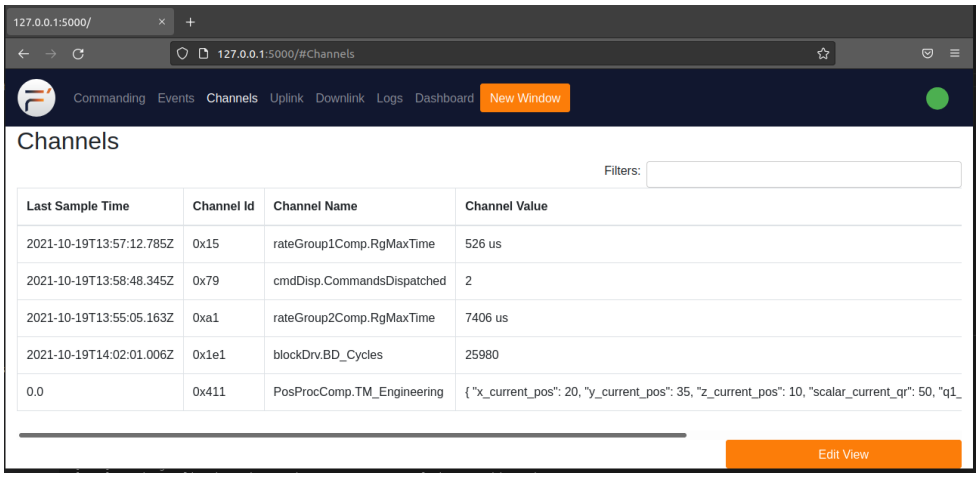
We can use the Ground System to perform some functional tests to ensure that the software system is properly implemented. All of the commands that can be sent to the flight software are accessible via the Commanding panel. We also record all commands that are sent, including the timestamp, identifier, mnemonics, and arguments.



When a command is sent, we can immediately switch to the Events panel to see if it was successfully executed.



Finally, we can access telemetry channel data by switching to the Channels panels.



Note: if the default layouts are inconvenient for us, we can load our own via the Dashboard panel.

8.1.4.5. Git repository

Our project fprime-poc was divided into four parts:

- The F´ framework “fprime”;
- Our internal framework “fprime-sodern”;
- A folder containing our mission specific components “MissionSpecificComp”;
- A "Top" folder containing the main and the XML defining the links between all components.

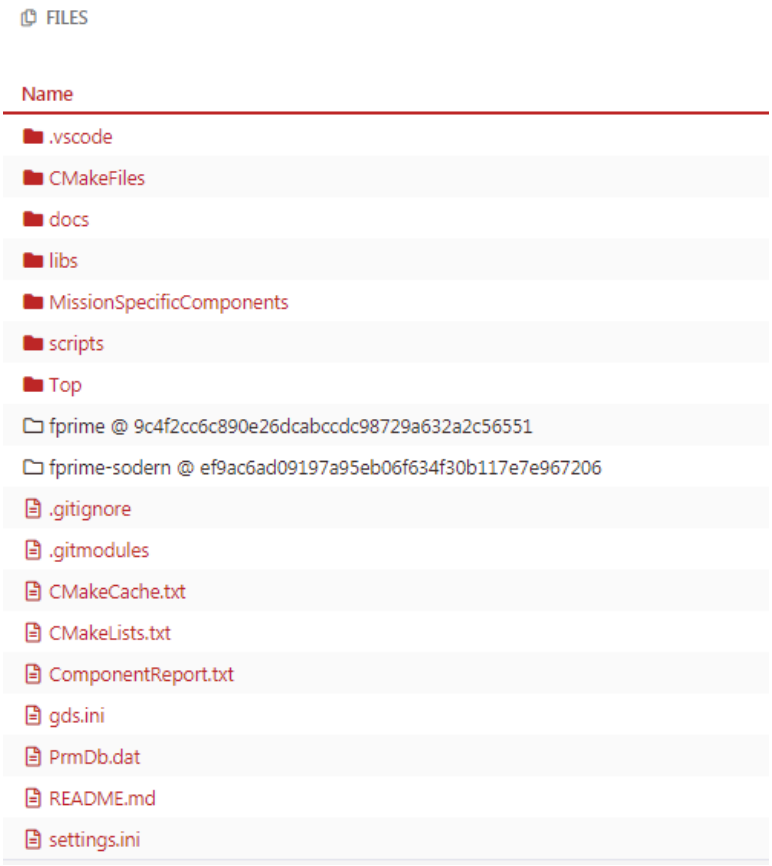


Figure 8-6: fprime-poc repository content

8.1.5. Synthesis

Resource Utilization

F Prime is written in C++11 in the "Embedded C++" style, with some coding rules inherited from the C language rules. These rules are intended to improve safety-critical dilemmas, readability and to allow modern development in C++.

We discovered that parameters specified in the XML definition of a component consume a significant amount of memory space. Adding a parameter increases the size of the program exponentially because of the way the parameters are processed. We did a quick measurement and found that each parameter took up about 350 bytes. We cannot use this parameter handler in the context of an embedded hardware platform with limited memory space. So we have to implement our own parameter manager by defining them manually and updating them via a memory patch.
This problem is known to the project and tracked in the GitHub issue tracker.

Interoperability

In terms of interoperability, the uplink and downlink message formats do not conform to space standards such as Space Packet Protocol or CCSDS SOIS EDS. Since the communication message format requirements of each mission may differ, it was decided to delegate the implementation of the project's serialization protocol.

We believe that compliance with the CCSDS SOIS EDS standard would benefit F Prime in the future by facilitating system interoperability.
It would also be interesting to propose an implementation of the space packet protocol without making it native.

From what we see professionally, both of the above standards are increasingly being specified by space industry prime contractors.
Some protocols, such as SPP, may not be necessary because they add overhead, and we may consider not using this protocol for some very low bandwidth missions.

Appropriateness recognizability

At the time of writing, there is no table that lists all of the projects and programs that have successfully implemented F´. We believe the following elements should be included in a table:

- Project/Program;
- SW Class (A through F);
- Prime Contractor;
- Cost Estimate;
- Operating System;
- Hardware Platform
- Launch date or TRL in the case of TRL < 9.

This type of table could reassure and encourage businesses to use F´ by allowing them to compare their project to a project already developed with F´ and find similarities.

Learnability

Despite the fact that F´ has a fairly complete documentation and several tutorials, we think that some "basic" features are not yet or not sufficiently documented. For example, the Rate Group section on the GitHub page is still marked as TODO despite being a frequently used feature, or a section on adding a library to a component that is not yet present.

The tutorials cover almost all the features of the framework, but we imagine that one or two tutorials could be added, for example to explain how to deliver F Prime as a simple library to be integrated into a customer's software. It would also be useful to have a tutorial explaining how to implement F Prime in the case of a TSP (Time Space Partitioning) architecture.

Popularity

In terms of popularity, the framework has gained international recognition as a result of Ingenuity media coverage, and the GitHub platform regularly publishes posts introducing the framework in order to grow the community. Despite a relevant number in the thousands of stars and forks, the community that shares on the GitHub forum lacks this dimension but is growing.

Perhaps the framework's spaceflight attribute implies that it is specific to the space domain, reducing the number of potential contributors who do not find application in their domain. Perhaps the association of the framework in GitPod or GitHub Codespace could facilitate contributions by providing a visual and quick demonstration of the framework without requiring complex user action.

Human-Computer Interaction

The "fprime-gds" graphical interface that communicates with the embedded software is quite complete and usable in a browser, which allows the F´ infrastructure to be fully containerized. It has an intuitive layout with different panels, and we can customize the dashboard to our needs.One drawback is that we were not able to send complex data types such as data structures; this is a known issue that has not yet been resolved in our development.

We note that there is a lack of data visualization to track telemetry in real time. The project was able to publish F Prime data to OpenMCT, but the lack of a command-sending solution prevented them from using it to replace fprime-gds. Since the project did not want to maintain a custom solution to integrate with OpenMCT for sending commands, they stayed with fprime-gds.

In our opinion the widespread use of CCSDS SOIS EDS could help move forward on these systems interfacing topics.

An experimental project to interface F´ with OpenMCT is available on GitHub, although it is not maintained, a stakeholder can study it to learn how to use OpenMCT.

Keeping fprime-gds, adding these visualizations would be simple using the Flask python framework or the Chart.js library.

Complexity

The afferent and efferent couplings of the components were extremely cumbersome in version 2.0.1. They were defined in the RefTopologyAi.xml file, which grew in size as the system became more complex, making analysis and inspection difficult. The project was aware of this problem, and in version 3.0.0, a tool called FPP was introduced to address it. This tool also allows the topology of the entire system or a subset of it to be modeled graphically.

Reusability

The reusability of the framework is reflected in the formalism of the XML-based component specification. It is the sole responsibility of the user to decompose their software system and capture the appropriate granularity of their components so that they can be reused on any framework-based mission. The user can create an internal sub-framework with multiple components shared by multiple projects, which can then be made available as a Git submodule of new projects. This creates a suite of ready-to-use applications for stakeholder IP issues.

Testability

The testability of the framework is reflected in the automatic generation of test component implementation templates, which include the GoogleTest unit test library. Components are decoupled from each other due to the separation of concerns, which allows the test component to be connected to the component under test, thus facilitating testing and enabling parallel testing. In addition, unit tests are compiled with the gcov code coverage analysis tool, and the results are saved in separate files for code coverage review. The framework also provides an integration test API to create an integration/acceptance level test on the target software.

Supportability / Serviceability

The fprime-util and fprime-gds tools are extremely efficient framework tools. They allow the developer to automate many tedious and human error-prone tasks. A possible improvement would be to allow the user to compile the source code directly from the tool without having to manually modify cmake by adding the -g option.

Adaptability

Concerning the portability of the framework on operating systems, the framework includes an abstraction layer for operating system "syscall" primitives such as thread management, interprocess communication (IPC), synchronizations, file management and timers. Currently, the supported operating systems are VxWorks, Linux, MacOS and baremetal (no OS). Although RTEMS support is not officially available, as it is POSIX compliant, it would be straightforward to add RTEMS support using the POSIX implementation. Attempts to port to FreeRTOS have been made, but it is still not supported because no hardware platform has provided a mature enough BSP. Porting will be simple to implement once such a platform is available.

Regarding its portability on processor, the framework can run on a wide range of processor architecture like x86, ARM, LEON3.

Regarding its portability to hardware platforms, especially for memory constrained ones; the framework has been ported to small platforms such as

- Teensy 3.8, a 32bit ARM-Cortex-M4 processor at 180MHz with 256KB of ROM and 64KB of RAM [90]
- ATMega128, an 8bit RISC-based AVR microcontroller with 128KB of programmable flash memory and 4KB of RAM [91].

Installability

While the framework installation is relatively simple, we may consider going through the process with the Poetry tool to further simplify the process and reduce the number of commands. For example, consider the command to activate the virtual environment.

8.2. NASA cFS

8.2.1. Overview

The core Flight System (cFS) is based on Goddard Space Flight Center (GSFC) heritage NASA Class “B” software and was initially applied to Class “B” and other less critical flight uses.
The cFS software has expanded its NASA mission footprint to both additional science missions and safety-critical human flight software (FSW) architectures. It has gained enormous popularity within the NASA software community, as well as the open source community, for its flight heritage and for its highly reusable and reconfigurable features [92].

NASA's Lunar Gateway program decided to use cFS as its software framework because of the cost and schedule success of many NASA flight projects that use cFS as their software framework [93]. This program funds the certification of the cFS product as Class A flight software. This class A certifiable cFS package, named Caelum or Draco, is expected to be made available to the open source community on GitHub.

The GitHub published package will be "certifiable," as "certified" software is software that has been verified and certified on a specific hardware and operating system, with all necessary certification artifacts resulting from the certification process. The GitHub package will include all necessary certification artifacts, test suites, and tools for formally verifying/certifying on specific hardware and operating systems.

There are numerous projects or programs that successfully use the cFS framework. cFS is used in a wide variety of software classifications and NASA centers. A few well-known projects that use cFS are listed below [92]:

Project/Program	OS	SW Class	NASA Center	HW	Launch
LRO – Lunar Reconnaissance Orbiter	VxWorks	B	GSFC	RAD750 (PowerPC 750 family)	2009
Magnetospheric Multiscale Mission	RTEMS	B	GSFC	Rad Hard Coldfire (5208)	2015
Dellingr	FreeRTOS	D	GSFC	Gomspace Nanomind A712d ARM7 RISC processor	2017
Parker Solar Probe (PSP)	RTEMS	B	APL	LEON3 UT699	2018
Orion Vision Processing Unit Artemis I (EM-1)	VxWorks	A	JSC	UT700 LEON 3FT	2021
Lunar Gateway	NC	A	-	-	2022
Orion Camera Controller Artemis II (EM-2)	Ubuntu- 64bits Linux	A	JSC	Intel i5 CPU (NUC)	2022
Orion Vision Processing Unit Artemis II (EM-2)	VxWorks	A	JSC	UT700 LEON 3FT	2022
Mars Ascent Vehicle (preliminary)	-		MSFC	Sphinx	2026 or 2031-

Figure 8-7: Some NASA Missions using cFS

8.2.2. Ecosystem

A very complete developer's guide is available in the cFE documentation [94] and provides major guidelines and common conventions for the development process of a software system using this framework. As a result, we will summarize the sections of our study that are relevant to us.

8.2.2.1. Taxonomy of a cFS project

The cFS repository is intended to serve as a collection of interconnected submodules. NASA has debated whether to implement cFS as a monorepository or as a multi-repository via git submodules, which they avoided by only considering the cFS repository as a sample bundle. The current plan for cFS is that it will not be used as is and will not exist as a sub-module in a project. Think of cFS as a kernel that projects take over and extend by adding extra functionality and acting as distributors.

This means that the mission project must adhere to the cFS reference mission tree implementation [95] .

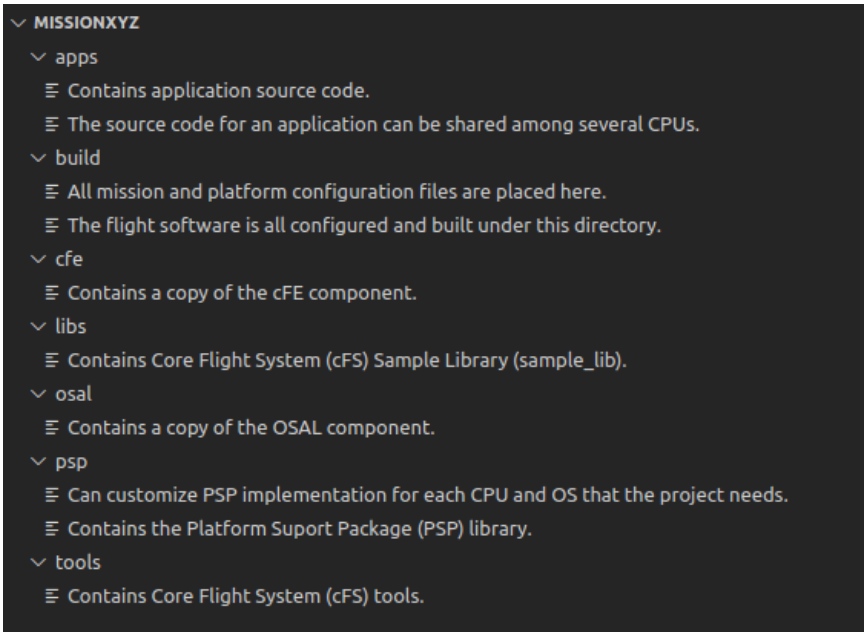


Figure 8-8: Taxonomy of a cFS project

8.2.2.2. cFS application design pattern

The best practice for creating a new application is to start from the reference “sample app”, whose compatibility with cFS evolutions is ensured. This is because a cFS application is defined with well-defined semantics where it sends and receives data in Service Data Units (SDUs) that respect the cFE interprocess communication protocol implemented by the cFE middleware Software Bus. Additionally, the "sample app" implements the interface with cFS libraries and table services.

The cFS application design pattern adheres to a framework-specific standardization while allowing the developer to implement the logic of the component.

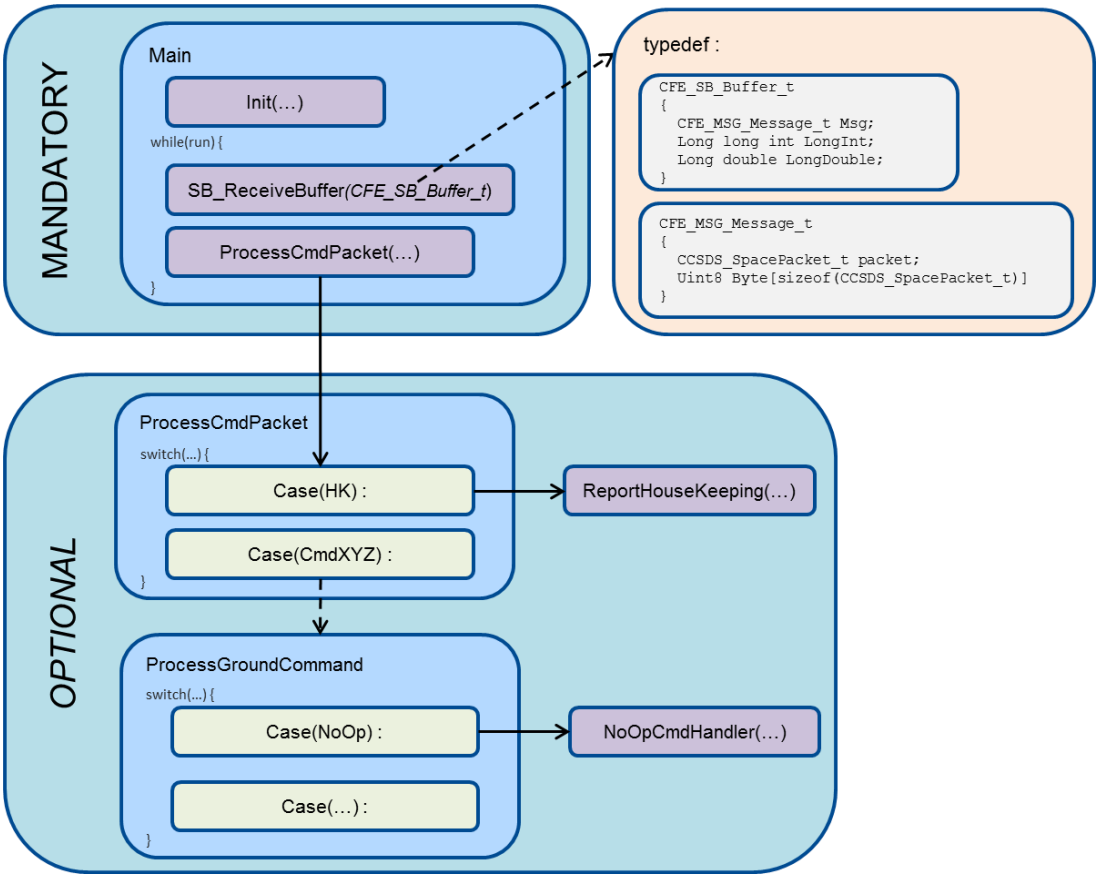


Figure 8-9: Design pattern of the sample_app application

The normalized part consists of:

- The application's entry point; it is through this function that the application is launched; it is also the only function callable from any other source module;
- In the case of specific initializations, an initialization function whose procedure can be customized by the developer;
- A main processing loop composed itself as long as the application has not been asked to be stopped, restarted, or reloaded:
    - A message reception function that is both blocking and infinitely waiting on the middleware service bus;
    - A function for processing received messages, the logic of which will be implemented by the developer;
- An application exit point that releases the application's various resources.

For the customizable part, we have:

- The content of the application's initialization function, where we can find:
    - Resetting the error or command counters;
    - Event initialization and recording;
    - Clearing of telemetry message buffers;
    - The creation and initialization of an input pipe to the service bus for the application to use in order to receive messages;
    - Subscriptions to specific messages consist of adding their identifier to the service bus pipe;
    - Adding the application's table to the table management service;

- The content of the message treatment function, in which we recover the message's identifier and invoke a developer-defined handler to determine the application's behavior after receiving the message.

NASA already offers several applications or interfaces, which are listed below:
- CS: Checksum;
- CF: CFDP;
- CI: Command Ingest;
- DS: Data Store;
- FM: File Manager;
- HK: Housekeeping;
- HS: Health and Safety;
- LC: Limit Checker;
- MD: Memory Dwell;
- MM: Memory Manager;
- SC: Stored Commands;
- SCA: Stored Command Absolute;
- SCH: Scheduler;
- TO: Telemetry Output;
- SIL: Simulink Interface Layer;
- ECI: External Code Interface.

8.2.2.3. cFS deployment

In order to generate the binary that will run on the CPU, the build chain must be started after all of the applications in the software system topology have been implemented. However, the mission's specific configurations must be specified before beginning the build chain. These must be centralized in a directory at the project's root with the suffix "_defs." Multiple directories with the suffix "_defs" can be used to access various operational environments such as debugging, hardware in the loop, simulation, and so on.

Just as we would when creating an application, we must retrieve the sample_def folder from the cfe submodule, which is located at the root of the cFS project. As a result, this directory is merely a starting point for the development team to tailor to the specific needs of the mission. It includes the following items:
- a file with the suffix "_mission_cfg.h" for global configuration parameters.
- The bus service's maximum message capacity, as well as the format of the UTC/TAI dates, can all be specified here.
- a file with the suffix "_platform_cfg.h" for each CPU for platform configuration parameters. It contains information such as the
  - CPU's identifier;
  - platform's endian indicator;
  - internal/external time source;
  - value of local clock rollover;
  - maximum number of shared libraries and applications;
  - default stack size for an application or task;
  - callback function in event of a CPU or floating-point exception;
  - maximum number of CPU resets performed prior to power-on reset;
  - activation/deactivation of event logging;
  - bus service's maximum unique message identifier, as well as unique pipes.
- a file with the suffix "_osconfig.h" for each CPU with CPU/Arch-specific configuration options for the operating system abstraction layer (OSAL). It contains information such as the maximum number of tasks, semaphores, and files that can be open at once.
- a startup script with the suffix "_cfe es startup.scr" for each CPU.

This file contains a list of all the applications and libraries that must be included in the software for a specific CPU. The type (application/library), the name of the component folder, the entry point, the name of the component, the application's priority and stack size, the component's loading address, and whether cFE should perform an action if the application throws an exception must all be specified for each component.

- a cmake file with the prefix "toolchain-" for each CPU to configure the cross toolchain to use the cross compiler for the target architecture.
- the file "targets.cmake," which describes the architecture and configuration of the target boards that will run the main flight software. It includes for each CPU:
    - a list of applications to build, statically or dynamically link, and install;
    - the tool chain configuration file;
    - the platform configuration file.

8.2.2.4. Command and telemetry messages in Ground System

Each telemetry message is defined in the *tools/GroundSystem/Subsystems/TlmGUI* folder in the form of "pages" containing the name and type of each telemetry field.

Each command is defined thanks to a utility "*CHeaderParser.py*" which will parse our header file ".h" in order to define the type of the structure to send as well as these fields.

8.2.2.5. cFS operation

We can build the flight software after we have completed all of the system topology. We will see some startup logs in the console after running the generated executable in the build folder.

The ground system is then started by running the python3 command and the GroundSystem.py file from the tools/cFS-GroundSystem folder. Once the ground system has been launched, we can interact with the flight software by using the various commands that are available.

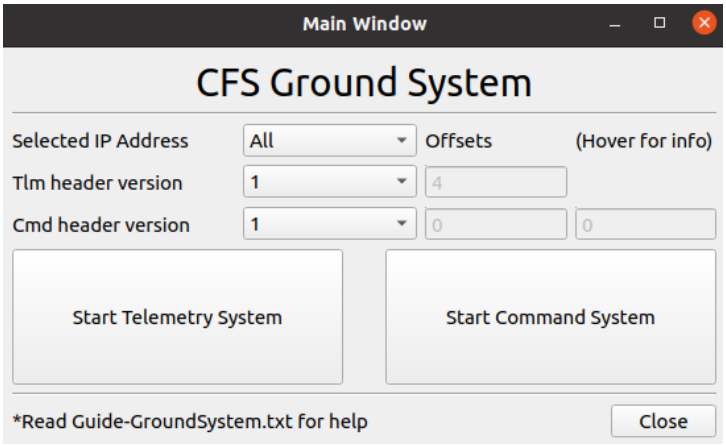


Figure 8-10: Global GUI of the GroundSystem

8.2.2.6. Testing and Continuous Integration in cFS

A third-party unit test library was considered but rejected due to the increased complexity it would add to the framework's unit tests. These tools are designed to meet the needs of a wide variety of usage contexts, and they tend to complicate and obscure the basic functionality required for unit testing [96].

As a result, cFS relies on the Unit Test (UT) Assert library for the unit testing phase, which provides a collection of utilities designed to make unit testing of framework applications easier. This library was designed to give the framework complete control over its functionality and complexity.

The cFS Test Framework – CTF [97] tool is currently used on the Lunar Gateway to develop and execute automated test scripts for mission requirements verification.
A graphical interface is used to configure the test scripts, which are then translated into JSON to contain test instructions, record, and report results. The tool employs a plugin-based architecture, allowing developers to extend it with new test instructions, external interfaces, and custom functionality.

In terms of Continuous Integration, cFS employs TravisCI to automate its entire testing process, which includes:

- LGTM in conjunction with CodeQL to identify vulnerabilities.
- LCov to obtain more information about unit tests, such as code coverage
- Cpp-Check to ensure the code's quality and readability.

cFS also employs other tools, the implementation of which and the results of which are not available to the open-source community. We can find the following:

- CodeSonar for identifying vulnerabilities and ensuring compliance with international standards
- Google AFL (American Fuzzing Loop) to perform fuzzing tests.

8.2.3. Architecture

The cFS architecture is built on multiple layers of abstractions that are designed to limit the impact of hardware or software changes while allowing for the insertion of mission-specific applications, allowing for significant formalized software reuse with associated cost, schedule, and software quality improvements.

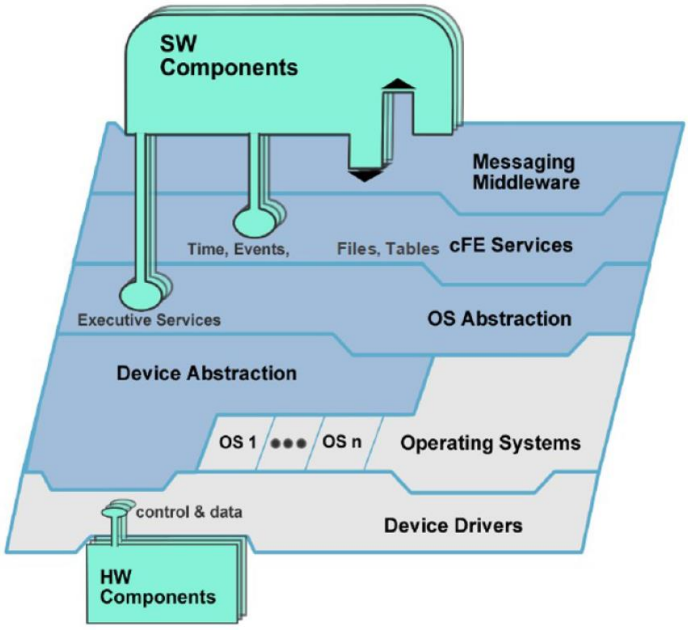


Figure 8-11: cFS Layered Service Architecture © NASA GSFC

In addition to explicitly reusable device abstraction, operating system abstraction, Core Flight Executive (CFE) services, and messaging middleware, the cFS architecture will:

- enable the creation of an ever-expending library of reusable or partially reusable mission applications tailored to the needs of individual user projects, and
- establish de facto internal interface practices that facilitate the rapid development of reusable applications across missions and projects.

The following figure depicts an alternate representation of the cFS FSW layers. From bottom to top, the cFS FSW has three layers:

- The abstraction library layer is made up of
    - Operating System Abstraction Layer (OSAL): a small software library that abstracts the flight software from the operating system.
    - Platform Support Package (PSP): software necessary to adapt the cFE kernel to a specific processor architecture and containing elements that do not fit in OSAL
- cFE core services, each of which includes an executable task:
    - Executive Services (ES): oversees the operation of the cFE kernel and cFS applications.
    - Event Services (EVS): provides a telemetry interface for asynchronously sending debug, information, or error messages.
    - Software Bus (SB): provides an inter-application messaging service that is portable.
    - Table Services (TBL): administers all cFS table images.
    - Time Services (TIME) - manages the spacecraft's internal time and distributes sound signals to interrogation applications.

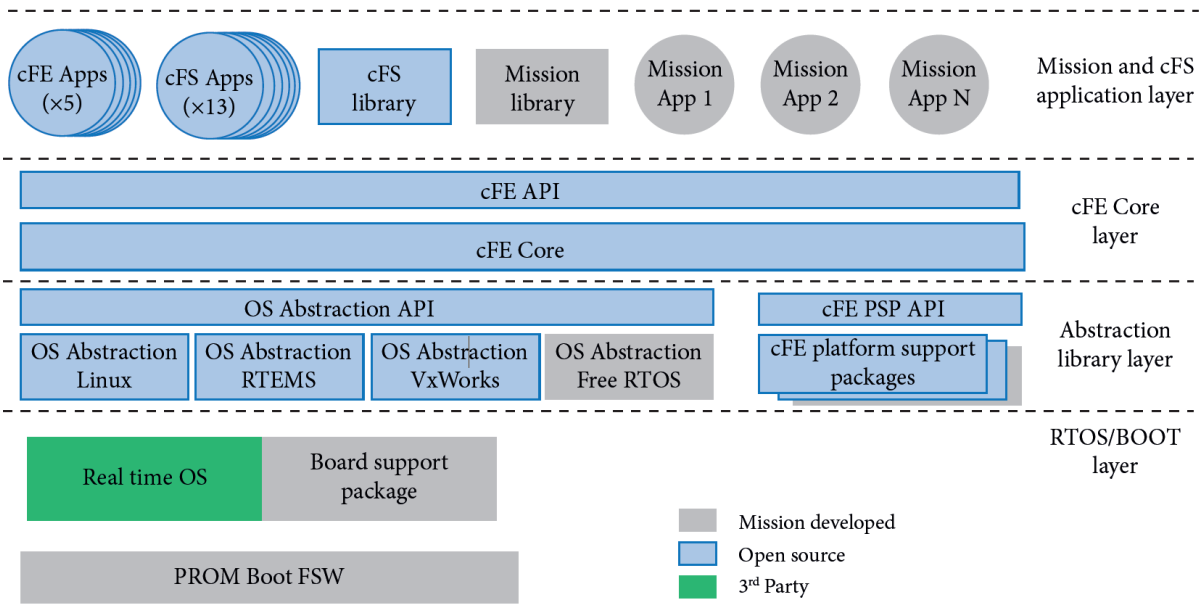


Figure 8-12: NASA cFS Software Layers and Components © NASA GSFC

The Executive Services is a layer that incorporates the OS Abstraction Layer (OSAL) API. The cFE has been designed to take advantage of this OS Abstraction Layer to improve its portability from one RTOS to the next. Since the cFE provides additional Executive Services that are not available with a standard RTOS, it stands between the OS API and the cFS Applications. The Executives Services allows providing:

- OS/Platform information: CPU identifier, number of ticks in a specific delta time;
- OS IPC: queues, binary semaphores, mutexes;
- interrupt handling;
- exceptions;
- memory utilities: pooling, read/write function, critical data store preserving data across processor resets, standard CRC calculation;
- file system utilities: device, directory, file functions;
- system log;
- software performance analysis.

The Software Bus service provides an API to cFS applications to ensure inter-application communication via message exchange. When an application is launched, it uses a subscription-based method to connect to this bus. Any application can send a message to the Software Bus, which will distribute it to all applications that have subscribed to receive the message identified by its identifier [94].

To receive messages, an application must first create its own pipes to the bus service, which are message queues that can only be read by the application that created them.
These message queues are limited by two parameters: the depth of the pipe and the maximum number of messages. The depth of a pipe is specified when it is created, limiting the number of messages that can be stored in it. When this limit is reached, any additional messages sent to it are rejected.

The message limit is set when you subscribe to a message, which limits the number of messages with the same message identifier that can be in a pipe at the same time. When this limit is reached, any additional messages of the same type are rejected.

The Event service notifies alerts on the downstream communication flow, which is the same as the telemetry messages flow. These alerts notify the user of occurrences or activities such as the receipt

of a command or the appearance of software or hardware errors. These messages are timestamped and assigned one of four severity levels: information, debug, error, or critical. This severity attribute assigned to an event serves as a filtering mechanism, with a threshold that can be adjusted per command to ensure that only messages of a particular severity are received.

The Table service allows managing an application's tables. A table is a database that uses contiguous memory space. This database is intended for storing configuration parameters that system engineers or flight software engineers believe will be changed during a mission, or for creating a mission-specific recurring product. We could look for real-time, calibration, and filtering threshold parameters, for example.

Each application table contains two images, one active and one passive. The memory area that is updated by commands is referred to as the passive image. The active image is the memory area that the application can access. After making the necessary changes, the table service can copy the memory area of the inactive image to the memory area of the active image. The table service also provides the option of managing the table with a single or double buffer. The benefit of double buffer management is that the application does not slow down while the table is being updated; however, it consumes twice as much memory space.

The Time Service allows each mission to define an epoch which is the time reference of a mission to which a number derived from seconds is added and which is necessary to determine an absolute time. Its API allows applications to access, convert, and manipulate the current time, as well as distribute and synchronize it. Time is represented by two 32-bit integers, the first representing the number of seconds and the second the number of subseconds. There are two time formats: international atomic time (TAI), which represents the number of seconds and subseconds since ground time, and coordinated universal time (UTC).

Aside from the core services, there is one that lacks its own executable task: the File service. It enables the creation, opening, and closing of files, as well as the reading, manipulation, and writing of data to them. A file is a collection of data, a textual or binary document, or an executable program. The File Manager (FM) application provides additional file, directory, and device commands for interacting with the embedded file system.

The "Decentralized System and Software Architecture" section of SOIS CCSDS refers to the cFE component of cFS [30]. cFE, like SOIS, employs a layered software architecture and includes a number of services that are similar to those specified in SOIS. Unlike SOIS, cFE is built on a software component architectural model, with a software bus serving as the primary data exchange mechanism between software components. A software driver connects hardware devices to software application components by connecting the hardware to the software bus.

8.2.4. Proof-Of-Concept

Our cFS proof-of-concept is divided into several main components for framework implementation:

- The cFS bundle which consists of cfe, osal, and psp elements;
- Our internal applications which are organized in a apps_sodern sub-module;
- Our mission specific applications, which are organized in a apps_mission folder;
- The sample_defs folder, which contains all of the files that comprise the flight software build chain;
- The tools and Python folders, which contain a variety of tools, including the GDS and a component generation script.

We discovered that a cFS distribution called Icarous provided some benefits that the cFS bundle did not, such as:

- Application/component auto-generation based on the sample application "sample_app";
- Automatic generation of the startup file cpu1-cfe-startup.scr, which allows the software to know which components to launch and how to launch them.

As a result, we decided to import these various tools into our prototype and improve them so that they could be integrated into the architecture of our prototype. The two applications that we imported and modified are located in the Python folder:

- *ConfigureApps.py*, which is called during the flight software build chain and allows for the generation of the startup file;
- And, *CreateApp.py*, which allows for the creation of the skeleton of a normalized application with a name specified by the developer.

8.2.4.1. Create application

The process of creating a new component is divided into several steps:

- Skeleton component auto-generation;
- Writing the component's internal logic;
- Adding the component to the compilation chain.

To generate the skeleton of a component automatically, we use the *CreateApp.py* script in the Python folder then Template. All we need to do is run the script with the name of our component as an argument. The script then generated a folder with the name of our component and all of the required source files. We then copied this folder into the "apps_mission" folder containing our applications.

We created the various functions in the ".c" file with the logic of the component so that it could render the service for which it was created. To compile and launch the component, we added the name of the module to the "APPS2RUN" variable in the apps.cmake file in the sample_defs folder.

In this PoC, we used the *CreateApp.py* file to create four modules:

- OHAcq;
- OHTempAcq;
- ImgProc;
- PosProc.

Each of these components is then mentionned in the *apps.cmake* file as follows:

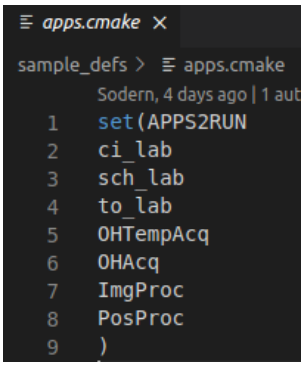


Figure 8-13: cFS-POC apps to run

8.2.4.2. Link application

The software bus connected each application to the others. To accomplish this, we created various message identifiers. On the software bus, for example, OHTempAcq sends a packet with the magic number 0x1899 every second. Because the OHAcq module is subscribed to the same magic number, it will receive the packet containing each OH's temperature (Optical Head). The diagram below depicts the configuration of these connections.

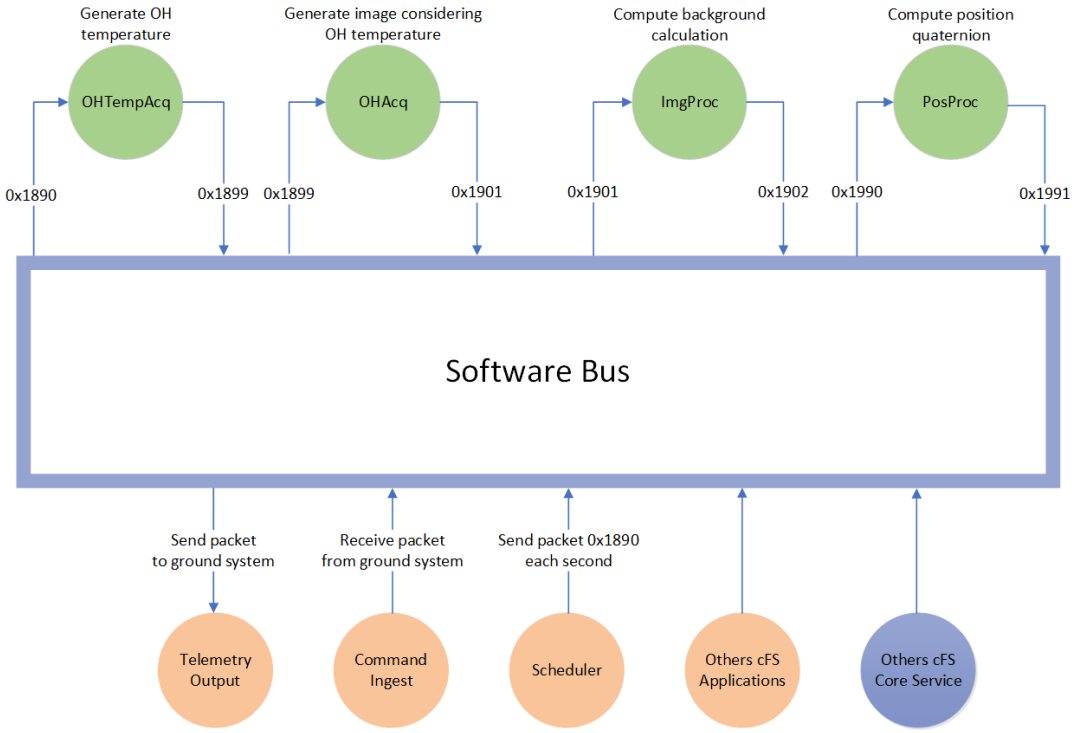


Figure 8-14: cFS-PoC topology

8.2.4.3. Launching the application

To start our cFS-PoC, we must first start the flight software, which we encapsulated in a build.sh script. The Ground System is then launched, with the command python3 and the file *GroundSystem.py* located in the folder tools/cFS-GroundSystem. Once the ground system has been launched, we can interact with the flight software using the various commands available.

By clicking on the different commands available on the right, a window opens in which we can enter our different parameters. On the left side, by clicking on the different buttons corresponding to each of the components we can find the telemetry received.

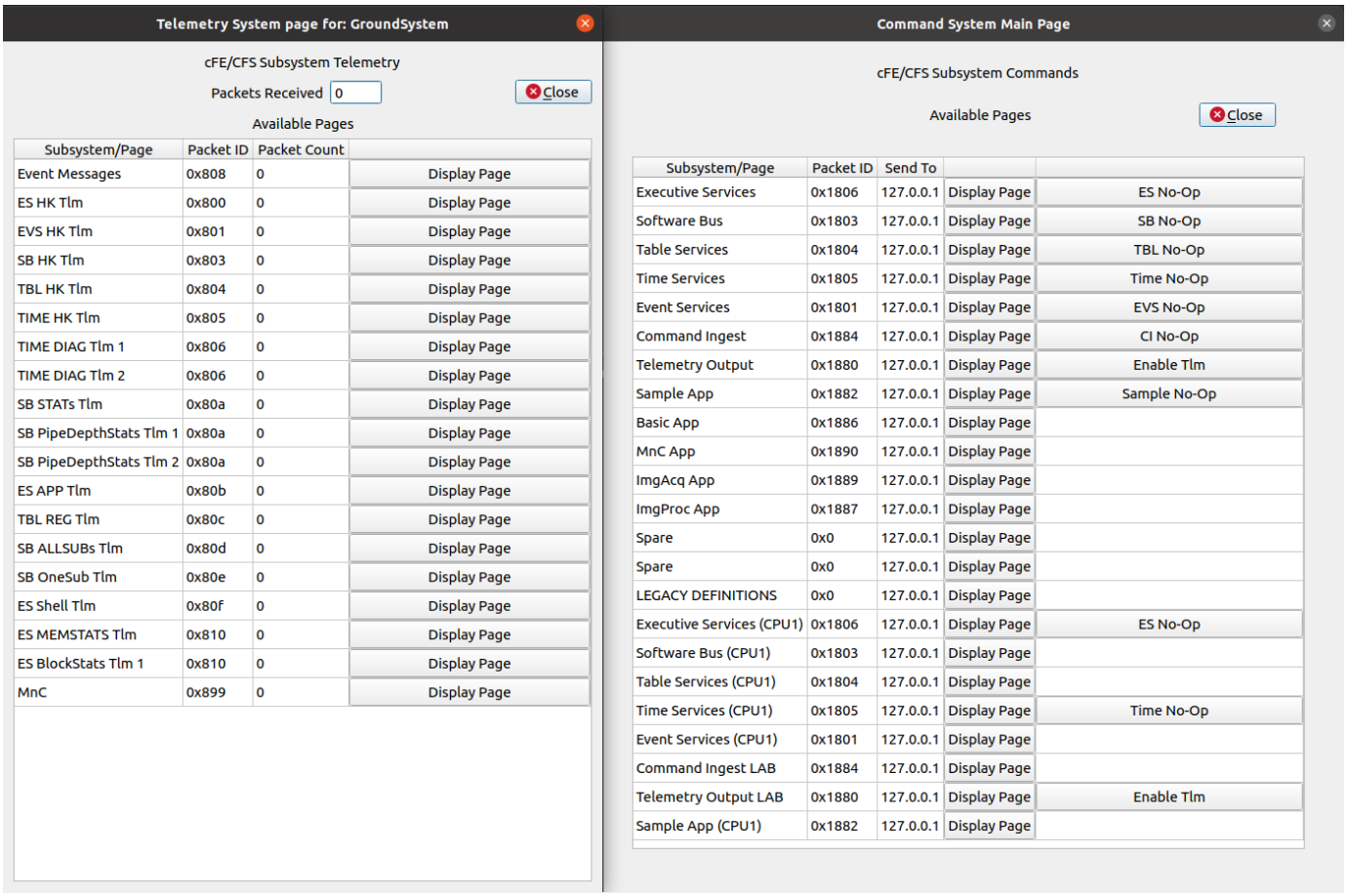


Figure 8-15: Cmd/Tlm GDS panels

8.2.4.4. Git Repository

The PoC project is archived in Tuleap under the project named cFS-PoC. It contains the source code for all of the applications and tools used in the proof-of-concept. The git submodule is used in conjunction with the cross-cutting applications "apps sodern" and the cFS distribution (cfe, osal and psp). Here's an example of the structure of the PoC repository:

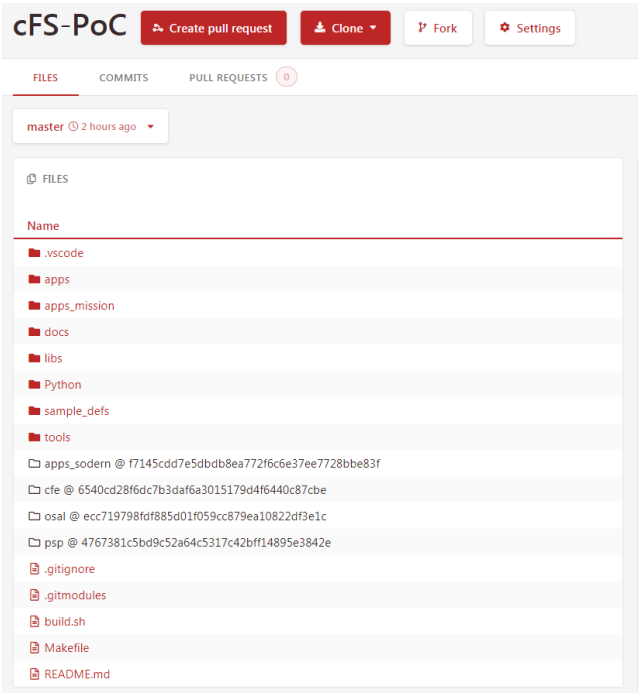


Figure 8-16: cFS-PoC archived in Tuleap

8.2.5. Synthesis

Interoperability

In terms of interoperability, the uplink and downlink message formats are consistent with the CCSDS Space Packet protocol. SEDS are currently being fully integrated into the framework.

Some SEDS files are present in cFS, but they are currently only a description of some applications; a future version that fully integrates them is expected. This does not appear to have been done before, especially since the use or non-use of SEDS between applications is decided at the mission level. In preparation for this release, the "cfe eds framework" project [98] shows how to integrate NASA EdsLib [99] with cFS and how EdsLib can be integrated into the cFS build chain. This project generates all platform-specific headers and messages from the SEDS files of each application. They can be a useful tool for learning SEDS unless they are used in the software system.

Popularity

Many NASA centers and prime contractors appear to be using the framework, but due to security policy, they do not appear to be able to openly discuss it in public forums. In our experience, there is a general lack of understanding of cFS among the general public. F´, a much newer NASA framework, has more than 20 times the number of stars on its GitHub repository than cFS. If cFS improves its communication about the project, it could have a much larger community, allowing it to receive feedback and new maintainers for the framework on an ongoing basis.

Human-Computer Interaction

The user experience was poor in terms of human-computer interaction. The framework's ground system is essentially just a development tool without dedicated maintenance resources. Its maintenance is limited to making it work well enough to meet basic needs, rather than using it as a production ground system. This lack of attention is due to the fact that NASA/GSFC has historically used its own proprietary ground system.
Currently, the ground system does not support floating numbers and complex types, image display, or real-time measurement tracking like mission control software, it is difficult to test our software. Thus, a new player attempting to obtain an operational framework and monitor system performance must expend significant effort to obtain a control screen.

There are no plans to improve the graphical tracking capabilities of the ground system to monitor the behavior of the embedded software. The philosophy being adopted is to move to a standardized data description and exchange format to facilitate the integration of more ground systems. The cFS could transform its Ground System application into a web application to allow their containerization and create a remote development environment (use of GitPod, GitHub CodeSpaces, and Visual Studio Code Remote Containers for example).

Portability

A significant drawback in terms of portability is that cFS does not support baremetal by default. The full capabilities of cFS are only supported when a file system is available (built-in or custom implementation). As it stands, it cannot be used in missions where memory is limited and an operating system is not available.

A typical use case for cFS is a software system that requires on-board maintenance, large data storage and operating/recording capabilities without communication, table management/configuration, etc. for which a file system is immensely beneficial. It is possible

to run cFS without a file system with significant effort and loss of capacity; the developer must adapt cFS accordingly so that it ignores abstraction layers that are unnecessary.

Testability

Concerning testability, we have two testing tools: one for unitary tests and one for verification.

The unitary tests are built with the UT-Assert library, which was created specifically for the framework, and are based on assertions and Stubs. On the Lunar Gateway, the CTF (cFS Test Framework) tool is currently used to develop and execute automated test and verification scripts for mission requirements verification. This open source tool, which is currently missing from the cFS project, should be added to Caelum.

An open source version of cFS certifiable class A, containing the entire test environment and allowing the software system to be certified on the user's hardware platform, should be released.

Modularity

In terms of modularity, the software architecture uses a plug-and-play approach in which each application can be dynamically loaded, connected or disconnected to the software bus service at run time without rebooting or rebuilding the software system. The software service bus serves as the cornerstone of the architecture allowing applications to communicate with each other through well-defined messages [100].

Roadmap / Governance

NASA appears to be more than willing to have private companies adopt their cFS framework because it is recommended on the Lunar Gateway and a CCSDS on cFS is being written to make its architecture a reference for space systems.

Despite the fact that the cFS architecture and implementation are among the best, we see a lack of tools in the cFS bundle that cover more phases of the space project life cycle.

To gain strength and impose itself on a global scale, we believe it should integrate or be inspired by the efforts of each distributor (OpenSatKit, NOS3, Icarous...) to rally the community around a common project and to be able to propose/work on disruptive technologies to implement.

If we want to integrate this type of functionality quickly, it might be useful to incorporate some features of cFS distributions into the bundle as we go.

Due to NASA's relationship with the US government, the release of cFS updates and GSFC applications are in a slow approval cycle of bureaucratic and legal hurdles, with the intervention of lawyers. In consequences the majorities of the GSFC application are out of date and cannot be launched with the most recent version of cFS. As a result, we must either use an old compatible version of cFS from one to two years ago or manually modify each component to fit the current version of cFS. Nonetheless, we believe that things are moving in the right direction; the development team is working hard to open up cFS as much as possible to open source.

8.3. ESA TASTE

8.3.1. Overview

Since 2005, the European Space Agency has been in charge of TASTE, or "The ASSERT Set of Tools for Engineering", a complex open source tool chain dedicated to the development of real-time embedded systems [101].

It is made up of a variety of elements that were created by various people and stakeholders in the space industry. The project is led by ESA, with contributions from Ellidis, ISAE Supaero, Neuropublic, UPM, N7Space, Pragmadev, and PoliMI. It is concerned with the modeling and deployment of distributed systems composed of various software and hardware components. It focuses on automating time-consuming, repetitive, and prone to human error tasks that make complex systems difficult to integrate, validate, and maintain.

TASTE is able to build systems from a high-level abstraction. It can create applications for the following architectures:

- x86 with Linux, Mac OS X, FreeBSD, RTEMS, and Windows;
- ARM with the GNAT runtime; and
- SPARC (LEON) with RTEMS and Open Ravenscar Kernel (ORK+).

The tool lacks a unified licensing system because it is made up of several components. All tools are completely free to use, and the vast majority are distributed under open-source licenses (GPL or LGPL) [102]. Because the runtime libraries and code of the TASTE code generators are part of the flight software, they are licensed in a way that allows them to be used freely, with no impact or licensing constraints of any kind. To put it another way, the multiple licenses have no bearing on the rights to operate and use the flight software.

8.3.2. Ecosystem

The TASTE tool is only available in a preconfigured virtual machine and runs only on the GNU/Linux distribution Debian 64-bit. It consists of the following components:

- a graphical interface for modeling/capturing the system's topology using the AADL language;
- an OpenGeode state machine editor that partially implements the SDL language;
- a certifiable ASN.1 compiler "ASN.1 SCC" dedicated to safety critical systems, generating encoders/decoders in Ada and C as well as their associated unit tests;
- a "glue layer" technology allowing heterogeneous code in terms of languages, allowing the developer to choose between C, Ada, Simulink, SDL, VHDL, and so on;
- Other tools that ensure a correct approach by construction, allowing simulations, regression, and monitoring tests, establish connections with system databases.

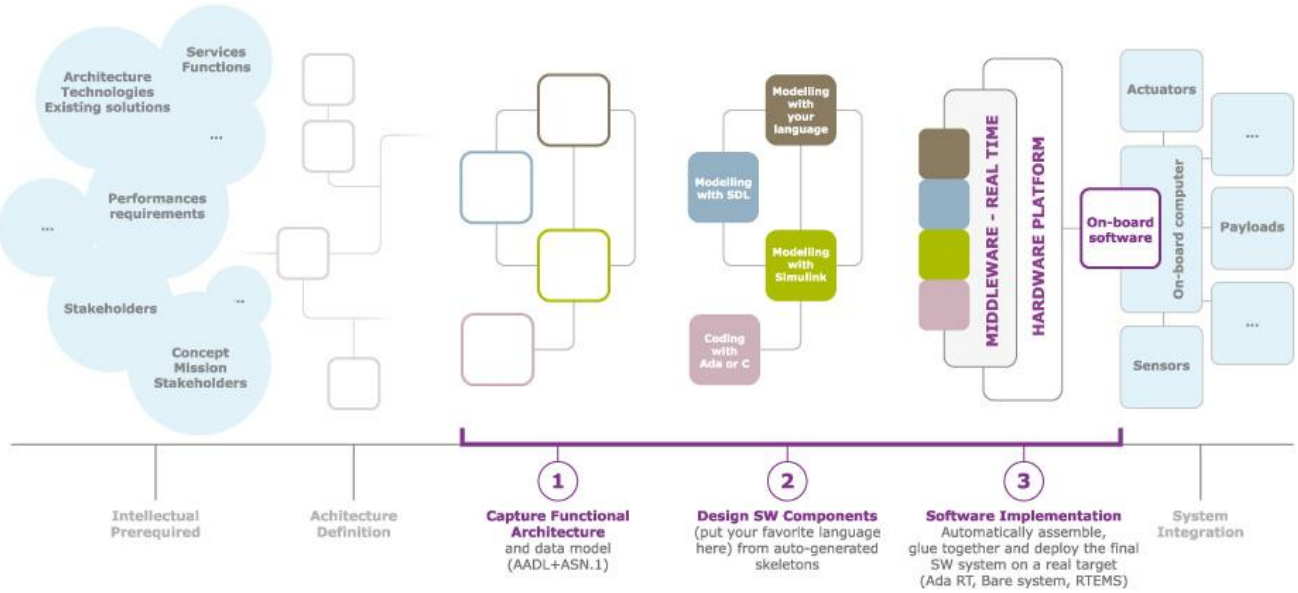


Figure 8-18: TASTE process © ESA/ESTEC

The graphical interface of the framework allows modeling the topology of the system within three different views: Data View, Interface View, and Deployment View.

8.3.2.1. Data View

The Data View panel enables us to define inter-component interfaces as well as interfaces between the space and ground systems by specifying the system's data types as well as the constraints associated with each of them.
The standardized ASN.1 language is used for data type specification.

We believe that using this specification format is inconvenient because it does not correspond to CCSDS-standard formats like the SOIS EDS or the XTCE.

Because this ASN.1 format is not currently widely used in our field of application, its use appears to limit the deployment of a TASTE-developed system to specific deployments, specifically technological mappings, such as on spacecraft with TASTE-developed embedded software.

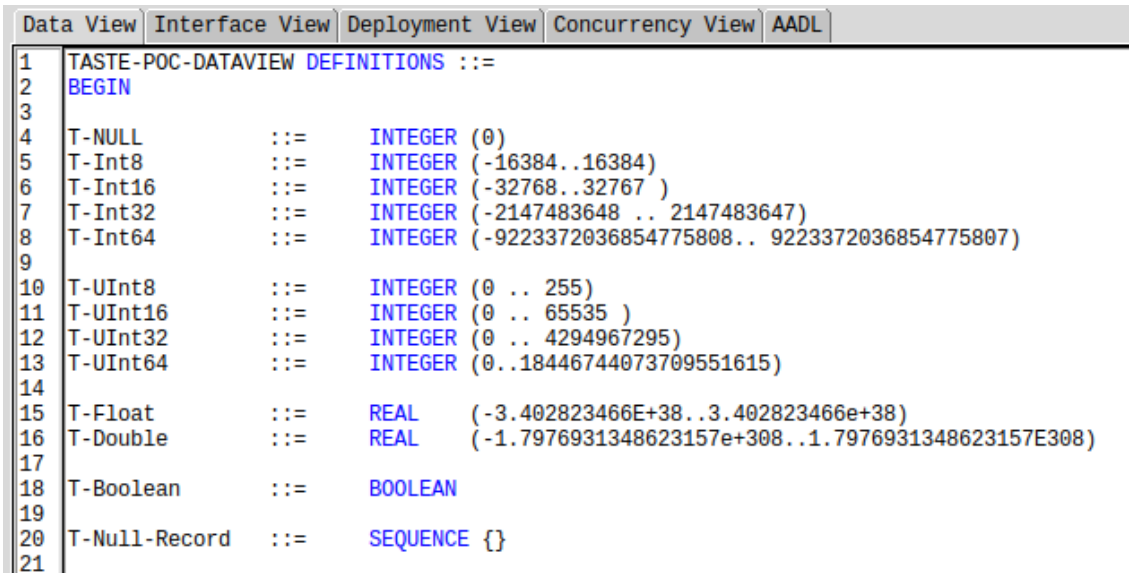


Figure 8-19: Data View panel

We quickly abandoned using the Data View window to edit our ASN.1 files due to the poor user experience. We can't view two ASN.1 files at the same time, and switching between them is difficult because the GUI constantly saves and checks the consistency of the file whenever we want to edit another. As a result, we only used Visual Studio Code to edit the ASN.1 files and double-check their consistency in the Data View.

8.3.2.2. Interface View

The Interface View is an abstraction that allows us to represent our system by reasoning about constituent blocks without needing access to all implementation aspects, especially source code.
The AADL language makes it simple to define "Function" components in the graphical editor and specify the programming language that will be used to implement the logic.

The system can handle a variety of languages: one component can be linked to another even if they don't use the same language, as long as it is one of the ones listed, such as C, ADA, SDL, or Simulink.

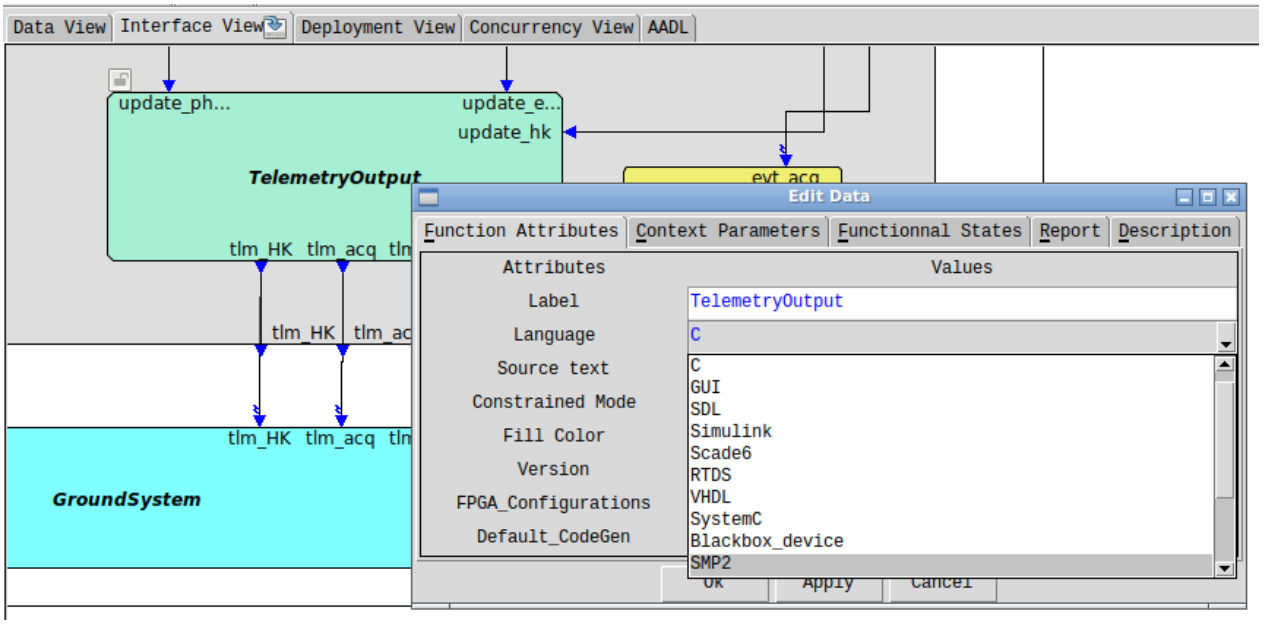


Figure 8-20: Component implementation language

The flow of data exchanged between two components is given by the direction of the parameter: input (Required Interface "RI") and output (Provided Interface "PI").
The interface specification contains the component's activation protocol, which comes in a variety of flavors:

- **Cyclic** is a no-parameter function that is used to perform cyclic activity on a component.
- **Sporadic** accepts an optional parameter and enables the component to be triggered on an episodic basis.
- **Protected** enables synchronous execution of the component by using semaphores to protect its interfaces, preventing concurrent execution of the component with other interfaces.
- **Unprotected** allows the component to be executed synchronously and can be executed concurrently by multiple interfaces.

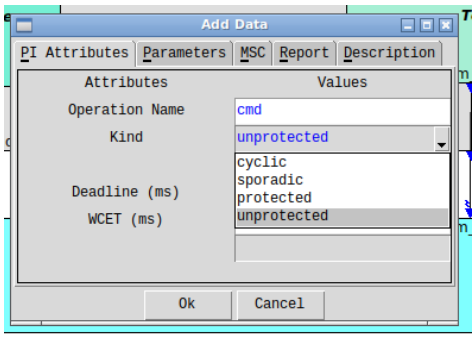


Figure 8-21: Component activation protocol

We can optionally add parameters after we've defined an interface. An interface, once defined, can have one or more parameters. Four characteristics define an interface parameter:

- A name found in the component's source code.
- A data type from the list of available options in the Data View.
- A Native, uPER, and ACN are all ASN.1 encoding types.
- An arrow pointing in one of two directions: IN or OUT.

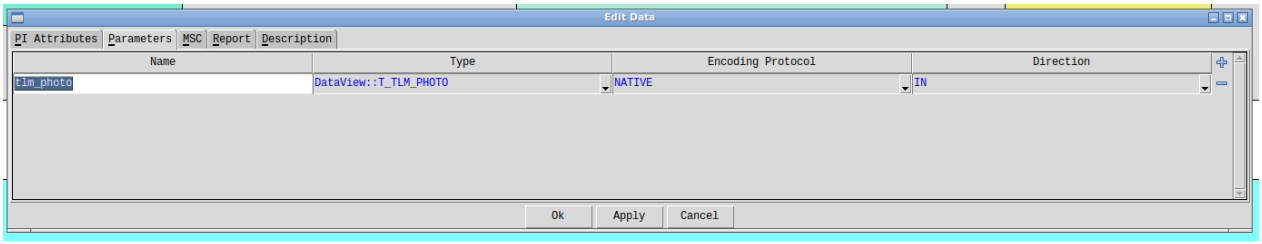


Figure 8-22: Specification of parameters

Once the topology of the system is defined, the graphical interface allows us to generate a skeleton code for each component. Then the development team just has to code the internal logic of each component in the placeholders of the functions that take as parameters those we had previously defined graphically.

It is possible to edit the source code of a component directly from the Interface View. By default, the editor launched is the Kate editor, unfortunately we have not seen the possibility to configure TASTE to specify another source code editor that we know in order to improve our productivity.

After defining the system's topology, the graphical interface allows us to generate skeleton code for each component. The development team is then only required to code the internal logic of each component in the function placeholders that take the graphically defined parameters.

The source code of a component can be edited directly from the Interface View. Unfortunately, we have not found a way to configure TASTE to launch another source code editor with which we are familiar in order to increase our productivity.

To work on the source code of a C component with the Visual Studio Code editor, modify the Makefile located at the root of the component folder and replace the command "kate" of the editor with the command "code," which launches Visual Studio Code.

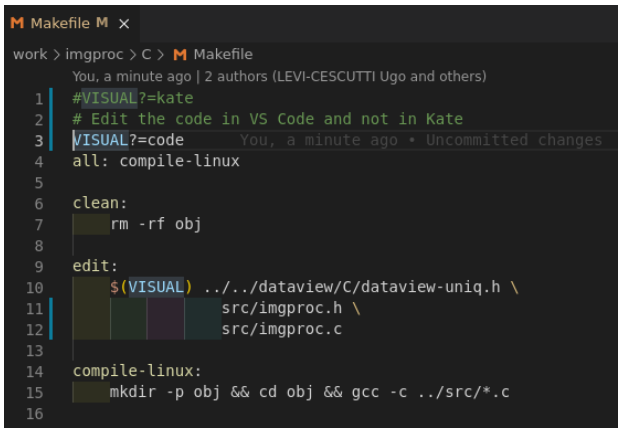


Figure 8-23: Change of source code editor in the Makefile

We quickly decided to open the entire project in Visual Studio Code because the process for each component was relatively time-consuming.

The code skeleton is generated using the file tree shown below:

- A src folder containing the code skeleton that we will modify with our own logic.
- A wrapper folder containing this component's glue code layer as well as the functions in charge of data transmission to another component equipped with an ASN.1 encoder/decoder.

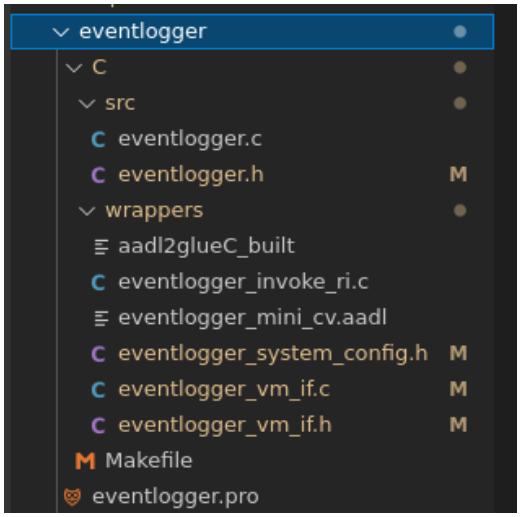


Figure 8-24 : File tree of the generated code skeleton

In the source file to be customized, we have a compilation unit, i.e. a ".C" file extension with a header file, i.e. an ".h" file extension.

The compilation unit contains only placeholders and looks like this:

- A startup function;
- One function per provided interface.

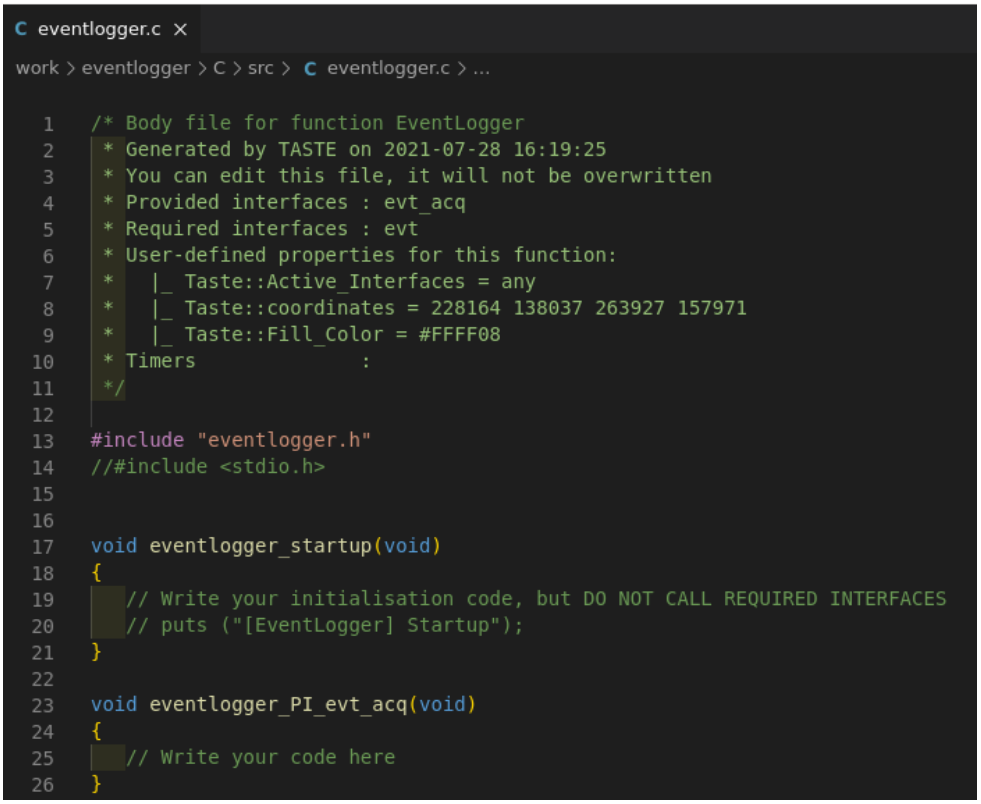


Figure 8-25: Generated c source file of a component

The header file is a little more special because the GUI overwrites it every time the code skeleton is generated. As a result, the developer cannot define macros or declare data in this file; it must only be regarded as belonging to the Interface View.

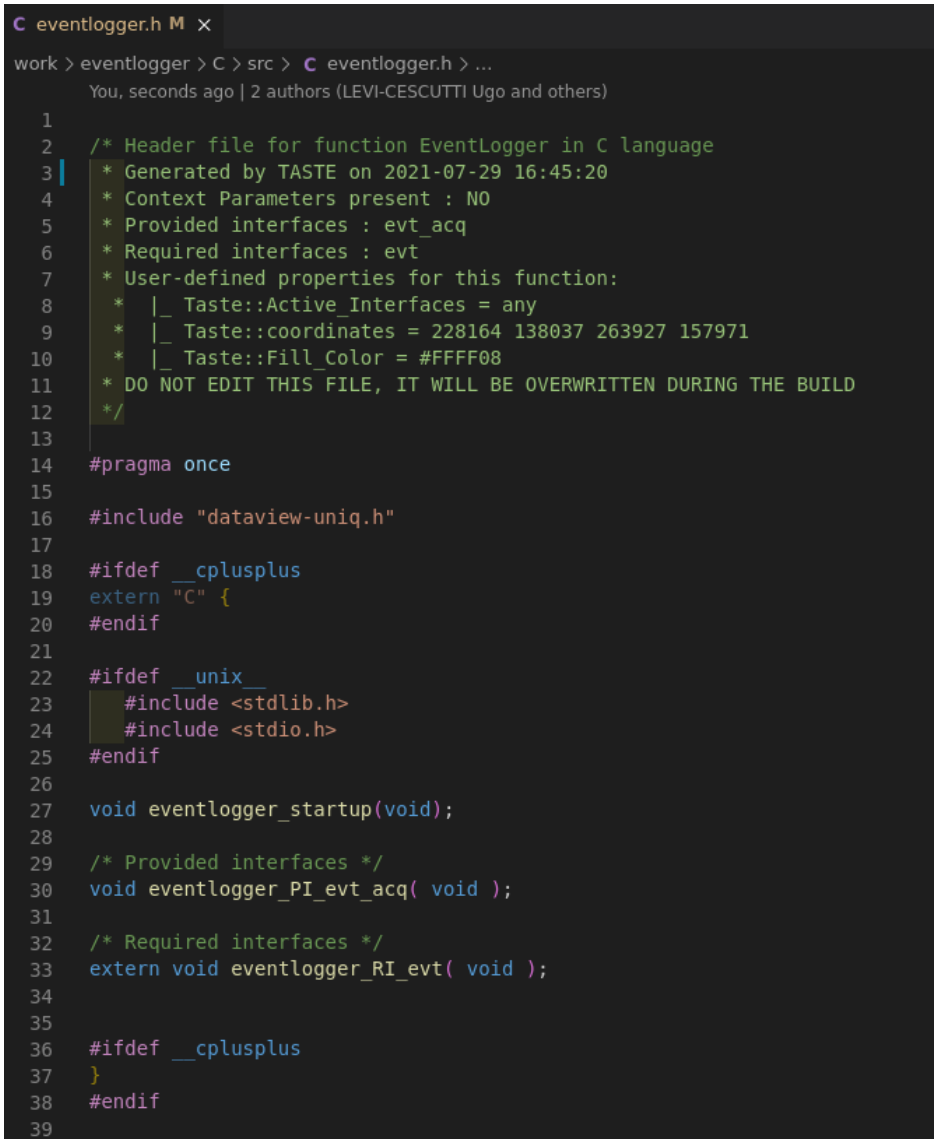


Figure 8-26: Generated header source file of a component

The tricky part of this code skeleton is that changing the number or name of an interface in a previously generated component overwrites the header file but not the C source file. Although it makes sense not to have deleted the C code because the developer may have written all of the complex logic for the component, we must completely rewrite the C file to account for the new interfaces defined in the header file.

As a result, we believe that adopting the NASA/JPL philosophy implemented on F Prime would be more interesting. That is, once the code skeleton is generated, the source files are generated with a ".tmpl" extension, which the developer must remove before writing to the file. This allows the developer to change the header file to define their macros while also making it easier to readjust the source file to component changes made in Interface View.

After we've coded the logic for each component, we can move on to the next step, which is concerned with the hardware platform on which our system is deployed.

8.3.2.3. Deployment View

The Deployment View enables to specify the hardware platform on which the software system is built. We can configure the Deployment View with three different types of elements:

- the processors that are supported
- the devices that are supported; and
- the data buses that are supported.

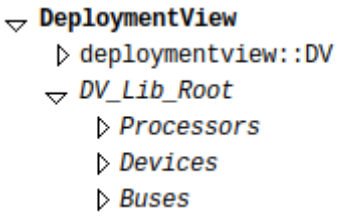


Figure 8-27: Deployment View elements

Among the available processors are LEON 2 and 3, x86, and Zynq.

Figure 8-28: List of supported processors

Among the available devices are LEON serial or Ethernet, generic serial or sockets.

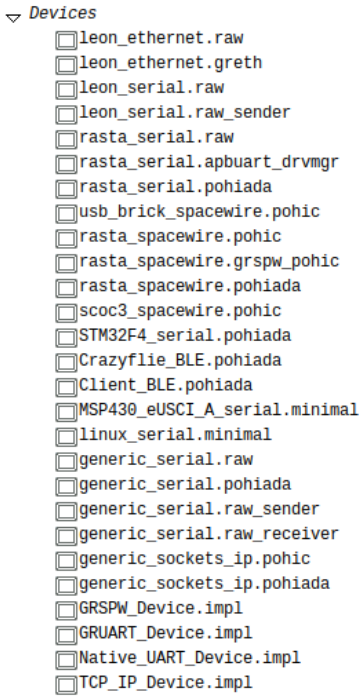


Figure 8-29: List of supported Devices

Among the available buses are Serial, Bluetooth or Spacewire.

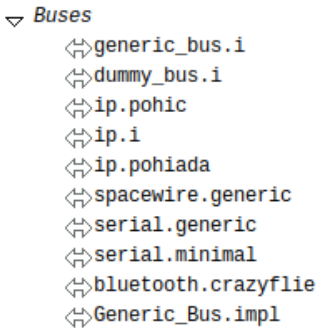


Figure 8-30: List of supported buses

In the Deployment View, we can model the entire hardware platform on which we want the software system to run.

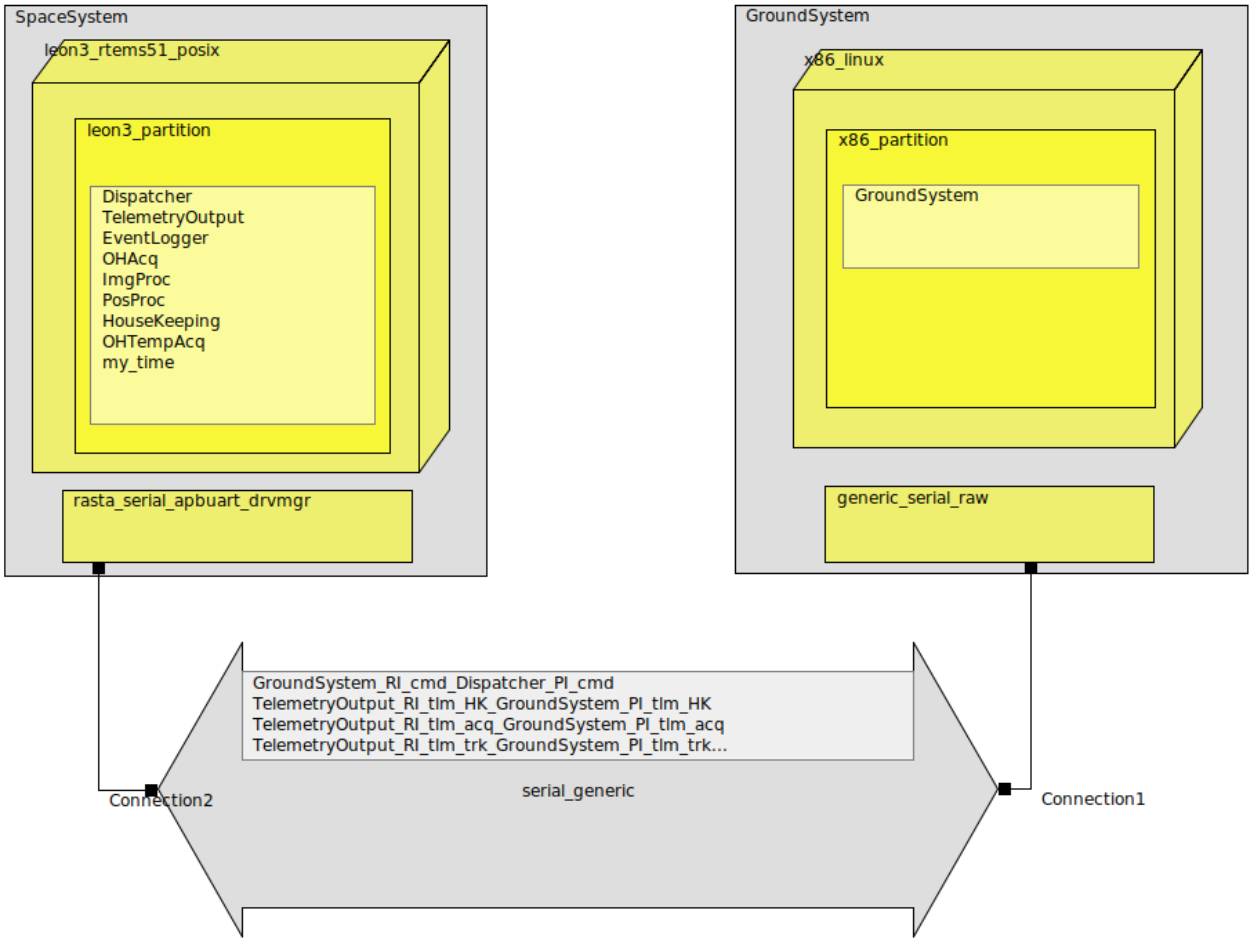


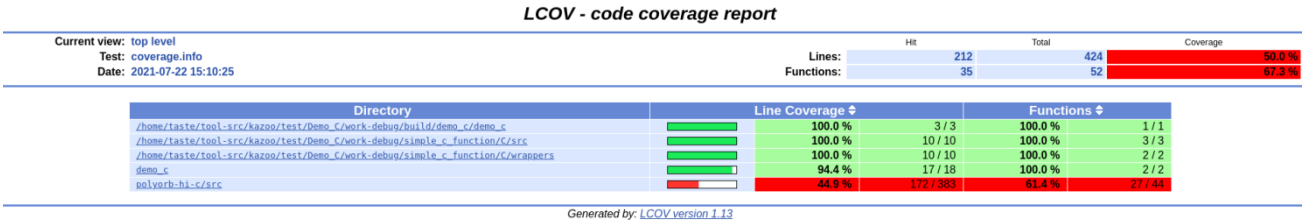
Figure 8-31: Heterogeneous deployment of a hardware platform

Please keep in mind that the above-mentioned deployment architecture was unable to be launched due to numerous implementation issues. Based on the available demonstration projects, this architecture should be implementable.

Once the deployment view has been defined, all that remains is to build the system by clicking the "Build the system" button, which saves the output binary to work/binaries.
After the build chain is complete, we can launch the binary of the Space System software in a first terminal using the command *taste-simulate-leon3*, which emulates a LEON 3 CPU, and the binary of the Ground System software in a second terminal using the traditional method.

8.3.2.4. Testing and Continuous Integration

The TASTE framework includes a test system that involves launching the program and observing which lines are executed over the course of 10 seconds. LCov provides a detailed summary of the number of lines and functions executed per file after the 10 seconds are up.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	212	424	50.0 %
Test:	coverage.info	Functions:	35	52	67.3 %
Date:	2021-07-22 15:10:25				

Directory	Line Coverage		Functions	
/home/taste/tool-src/kazoo/test/Demo_C/work-debug/build/demo_c/demo_c	100.0 %	3 / 3	100.0 %	1 / 1
/home/taste/tool-src/kazoo/test/Demo_C/work-debug/simple_c_function/C/src	100.0 %	10 / 10	100.0 %	3 / 3
/home/taste/tool-src/kazoo/test/Demo_C/work-debug/simple_c_function/C/wrappers	100.0 %	10 / 10	100.0 %	2 / 2
demo_c	94.4 %	17 / 18	100.0 %	2 / 2
polyorb-hi-c/src	44.9 %	172 / 383	61.4 %	27 / 44

Generated by: LCOV version 1.13

Figure 8-32: TASTE - code coverage

On their GitLab directory, TASTE also has a CI. This CI only checks if the project is built at each push; if the job fails, the commit is not pushed. TASTE also provides a LEON 2/3 CPU emulator, allowing to test the software without putting it on a board.

8.3.3. Architecture

The TASTE tool divides an application into four layers: application, glue, middleware, and OS with BSP support.

8.3.3.1. Application Layer

The Application layer corresponds to the applications/functions that the user has created and filled using a programming language (Ada or C) or application models (SDL or Simulink).
These applications are constructed using a code skeleton generated automatically by the previously mentioned Interface View.

8.3.3.2. The Glue Layer

The glue layer is generated automatically by the Kazoo tool and is used to connect the application layer code to lower level software components. This layer contains resources (tasks, buffers, and data) that enable system components to communicate with one another (other components on the same hardware platform or on a remote platform). It also makes use of runtime data structure encoders and decoders to convert data between ASN.1 and C/Ada/SCADE/Simulink/RTDS, enabling any function to communicate with any other.

8.3.3.3. The Middleware

The middleware layer connects the glue layer to the underlying operating system and BSP functions. This layer, when compared to cFS, combines the OSAL and PSP layers.
The layer modifies the application and glue layer's code to allow it to be compiled and executed in the target environment. It serves as an abstraction layer for a wide range of operating systems (Linux, RTEMS, Ada Runtime).

8.3.3.4. The OS with BSP support

The operating system with the Board Support Package is the intended execution environment (BSP). TASTE adjusts to the underlying supported operating systems (Linux and RTEMS 4.8/4.10) as well as the functions of the BSP.

8.3.4. Proof-Of-Concept

The design of the Proo-Of-Concept topology was completed relatively quickly in comparison to other frameworks because the TASTE tool is an MBSE tool.

8.3.4.1. Data View

All types used as interfaces for our components in the Data view have been specified in ASN.1. We decided to use a CHOICE tag to group the commands transiting on the ascending data flow from GroundSystem to SpaceSystem.
This keyword allows the source code to define a single reception buffer to store received commands, similar to a union in C language.

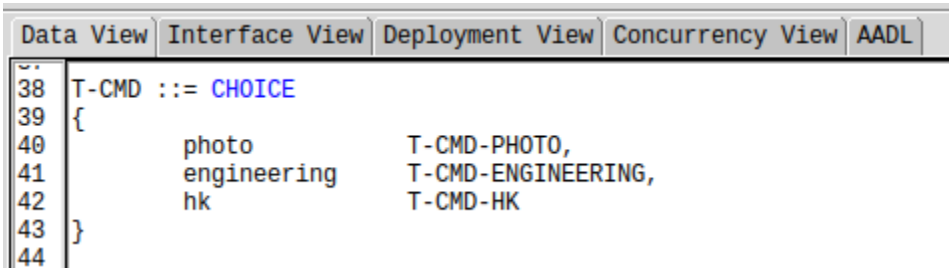


Figure 8-33: Command format in ASN.1

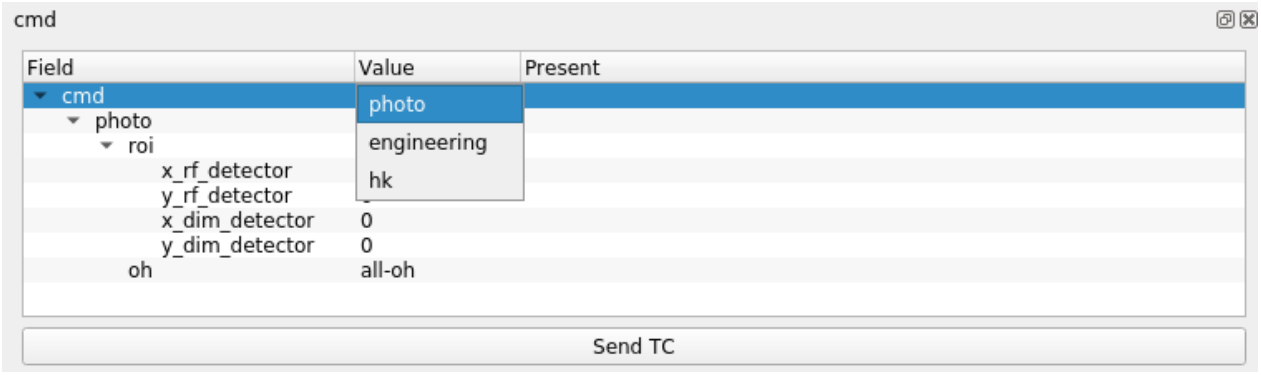


Figure 8-34: Displaying commands in TASTE

Another interesting feature of this keyword is that we have a single command panel in the GroundSystem GUI from which we can select the type of command, and the panel will update itself based on its content. The graphical display of commands with a CHOICE type allows us to select the type of command we want to send, and the display panel adapts based on the command's content.

We decided against sending telemetry from the space system to the ground system via a single channel. Placing them in a CHOICE tag would have been the most elegant solution because it would have resulted in a single send buffer for all telemetry channels, saving memory space.

We could also have combined the telemetry in the same SEQUENCE tag to create a memory buffer that contained each send buffer associated with each telemetry channel. To make the PoC work with TASTE, we did not try to optimize the architecture with ASN.1 types. Graphically, the telemetry display looks like this:

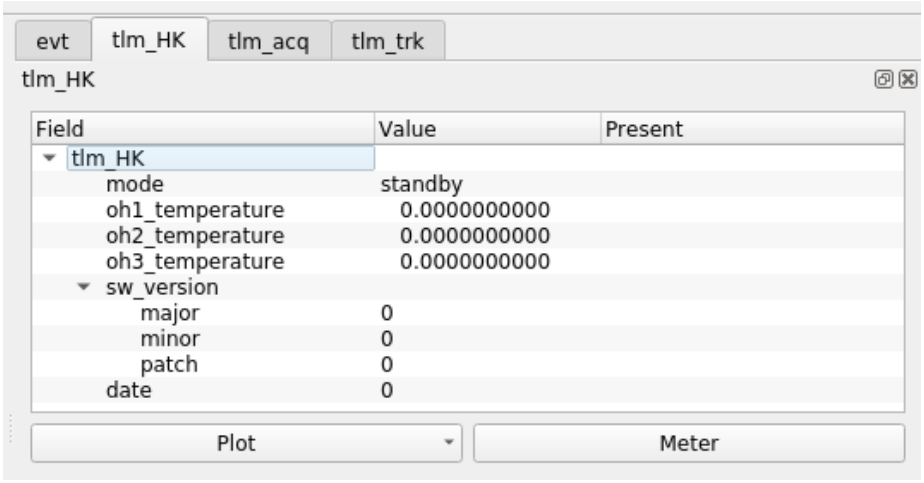


Figure 8-35: Displaying telemetry channels in TASTE

It should be noted, however, that we were forced to limit the size of the image generated by the embedded software. We defined the type of buffer that would hold the image's pixels in ASN.1, but this caused the software to crash. The cause has not been determined, but memory borrowing is most likely because we had to reduce the size from 2048*2048 to 30*30 pixels to avoid the crash.

8.3.4.2. Interface View

The View interface provided us with all of the tools we needed to model the topology described at the beginning of the report. When we model the topology, we get the following result:

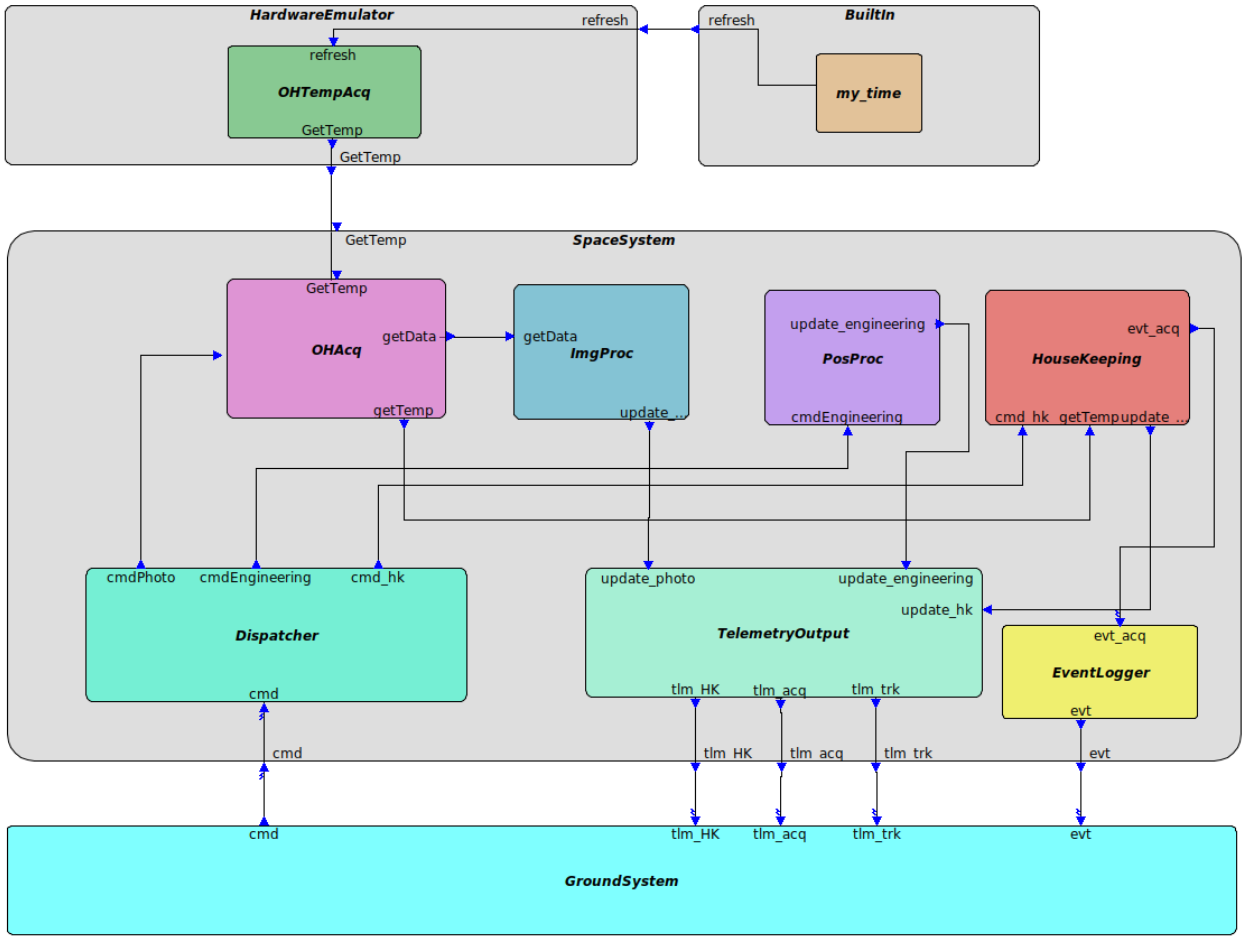


Figure 8-36: Rendering of the PoC topology with the TASTE tool

Do not duplicate the system model based solely on visible components and interfaces.
To avoid overcrowding the overall system view, the closer we zoom in, the less detail we get.
We must clone the project's Git repository in Tuleap in order to fully implement the aforementioned topology.

The code skeleton was then generated to code each component's internal logic.
During this implementation, we discovered that, despite their names, all ASN.1 basic types generated by the ASN.1 SCC compiler are 64 bits. An *uint64_t*, for example, is hidden behind the type *asn1SccT_UInt8*.

8.3.4.3. Deployment View

Regarding the deployment platform, we simply installed the PoC on a Debian virtual machine running Linux and the x86 hardware architecture that corresponded to our native machine.

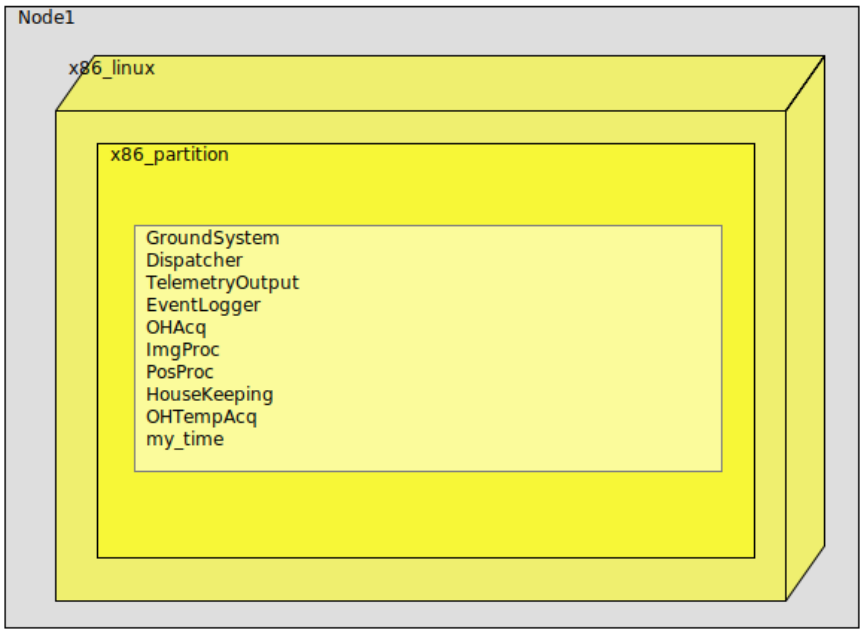


Figure 8-37: TASTE PoC deployment

We wanted to deploy on an RTEMS/LEON 3 emulation, but we couldn't get the LEON 3 node to communicate with the Linux node, even though both work independently and with assistance from the ESA development team.

8.3.4.4. Launching the application

Once the two systems, the Ground System and the Space System, have been built, we can launch them using the script *run_x86_partition*. Once the two systems are launched, we will be able to communicate with the Space System via the Ground System:

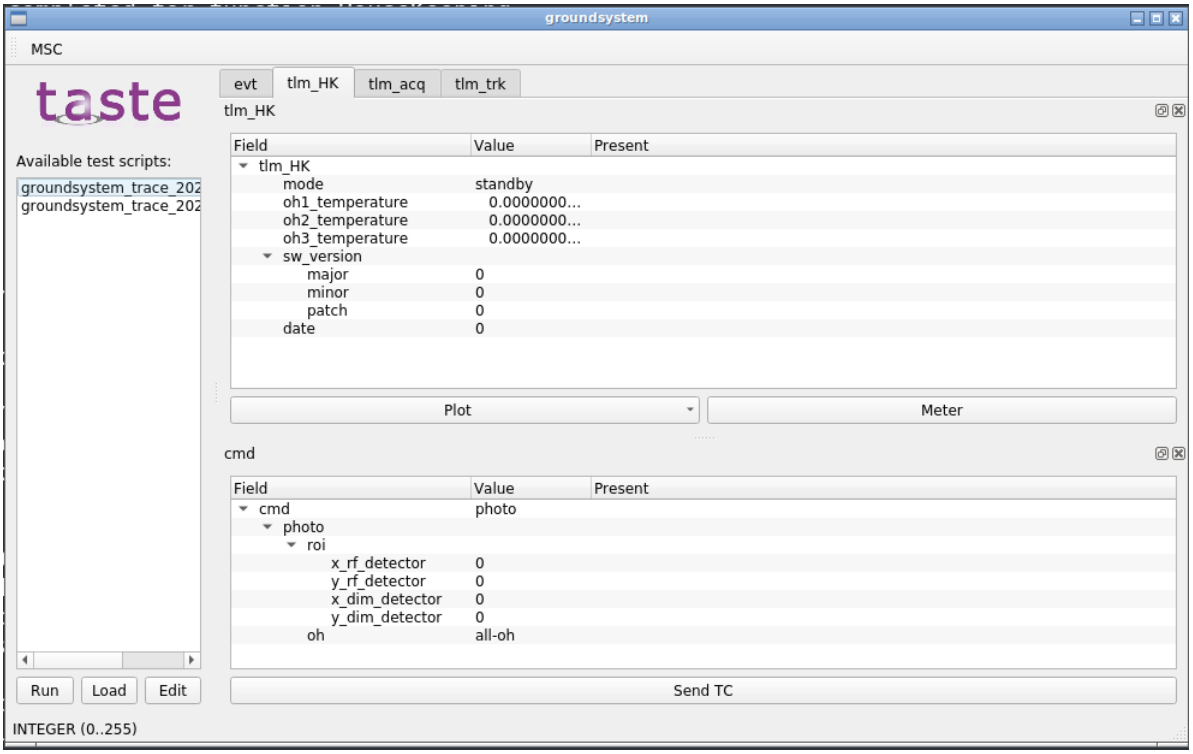


Figure 8-38: Global GUI of the GroundSystem

The graphical user interface allows for the creation of graphs that show the temporal evolution of data from a telemetry channel selected by the user. We have the option of integrating the plot directly into the main window or detaching it from it.

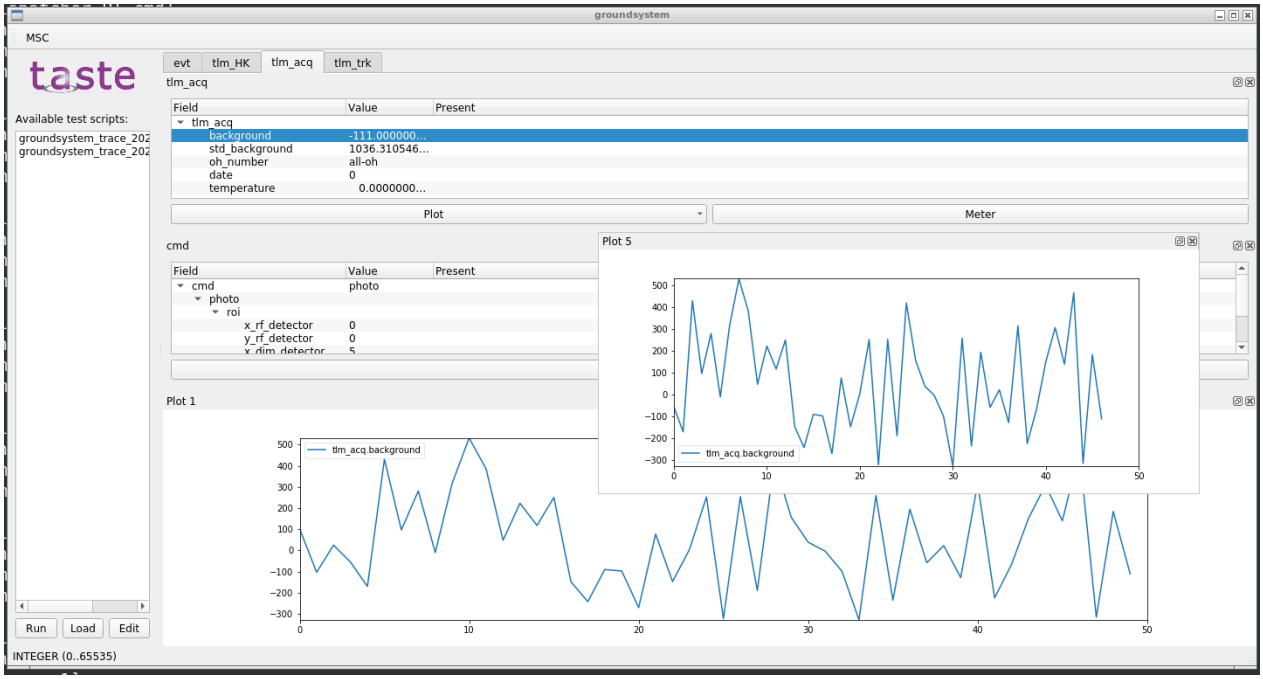


Figure 8-39: Plotting telemetry message values

In the image above, for example, we've decided to plot the temporal evolution of two fields from the acquisition telemetry message:

- the Background in the main window;
- And the StdBackground in a detached window.

In addition to the graphical interface, MSC (Message Sequence Charts) streaming is available. It displays the command and telemetry message sequences exchanged between the two systems/nodes during system operation:

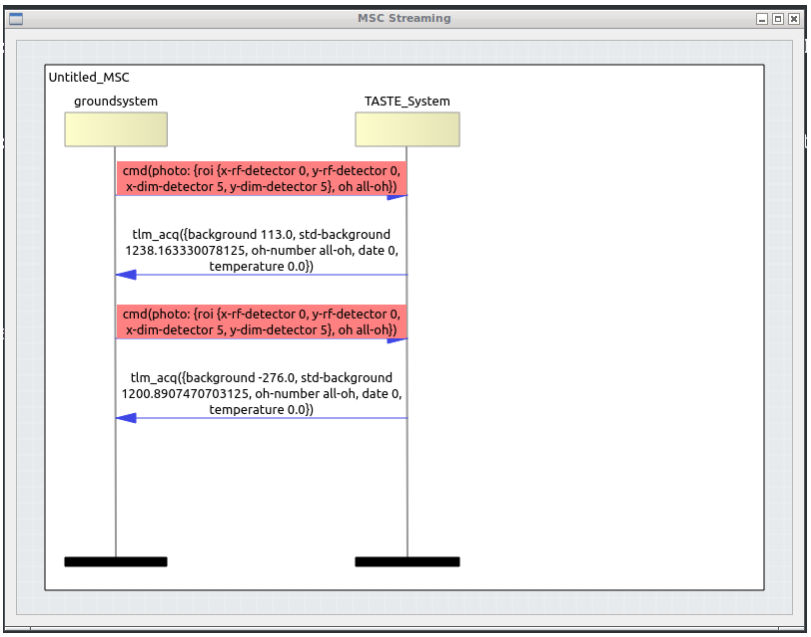


Figure 8-40: TASTE - MSC Streaming

In theory, we can use this MSC recorder to capture a scenario, which can then be refined in the TASTE MSC editor so that we can replay the scenario from the MSC files. The MSCs are converted to Python scripts, which we then modify to define validation, regression, and integration testing scenarios.

This is only a synopsis of the theory covered in the TASTE Wiki tutorial "Advanced testing with MSC and Python scripts." We encountered bugs when launching the MSC window, editing an MSC script with the "Edit" button, or launching the MSC Editor. We didn't want to implement this feature because of the numerous bugs in it because it would significantly slow down the development of the PoC.

8.3.4.5. Git Repository

The PoC project is archived on Tuleap under the project name *taste-poc*. It includes the source files needed for the TASTE tool to work properly. We did not create a gitsubmodule for taste-setup because the TASTE tool is already included in the Debian virtual machine, and including taste-setup in the PoC project is not required.

The repository structure of the PoC is depicted in the image below:

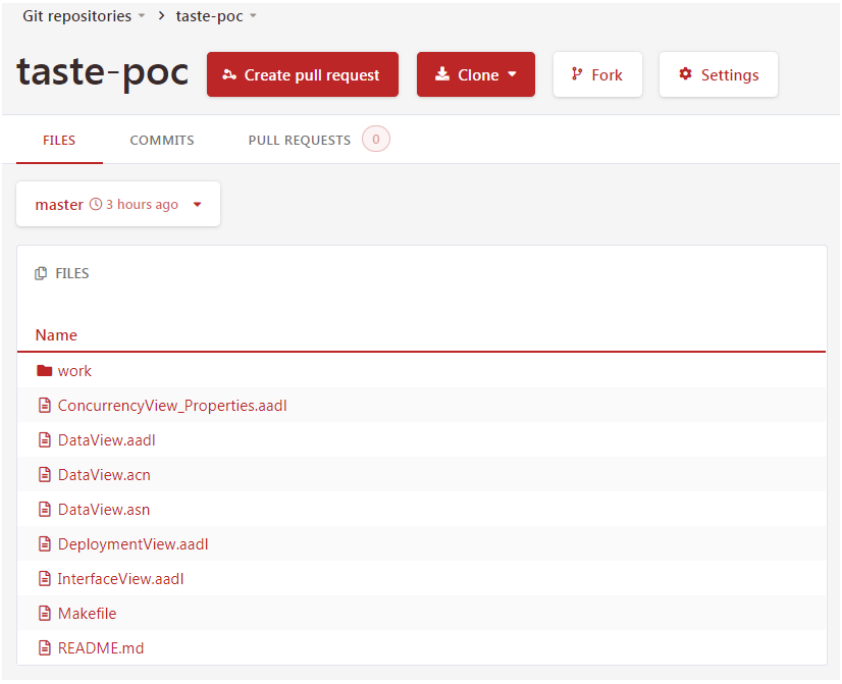


Figure 8-41: taste-poc repository

8.3.5. Synthesis

Resource Utilization

We discovered a problem with the ASN.1 SCC compiler's memory footprint. Indeed, whether an 8-bit ASN.1 or a 32-bit ASN.2 occupier type is defined, the resulting code will be 64-bit on a 64-bit platform or 32-bit on a 32-bit platform. As a result, if we define an 8-bit pixel buffer to store an image, each element in our case will have a size of 64 bits multiplied by 8. This, we believe, is why we had to reduce the size of our images from 2048*2048 to 10*10.

Interoperability

The ASN.1 specification format, as shown in the Data View presentation, is inconvenient for us because it does not correspond to CCSDS standard formats such as SOIS EDS or XTCE. Because this ASN.1 format is not widely used in our application domain, its use appears to limit the deployment of a TASTE-developed system to specific deployments, such as on spacecraft with TASTE-developed onboard software.

Accessibility

TASTE would allow various teams to share a workspace, such as the system team, flight software team, and digital electronics team. We can use the tool to develop our system from various angles.

The Data View allows defining the types that will be used to define the interactions in ASN.1 between two components.

The Interface View employs an MBSE approach, allowing modeling the system's topology without worrying about how to implement it in a programming language. Using code skeletons, it is possible to make a system's implementation in C or Ada transparent, and improvements to generate code skeletons for C++ or Rust are also possible.

Using the Deployment View, we can specify the software and hardware platforms on which the software topology will be deployed. Partitions, operating systems, and processors can all be modified.

Learnability

The TASTE wiki and learning resources should be improved and updated as a whole. Some procedures haven't been updated in ten years, and many screenshots and explanations are no longer relevant. Also, for each tutorial in kazoo, a README.md would be required to quickly describe what the particularity of this tutorial is and how it works or how to reuse it in the project.

Popularity

We found that gathering information was difficult and time consuming, which we attribute to the platform on which it is hosted. TASTE is hosted on GitLab, a platform for private projects rather than public/open source projects. For public projects that want to reach a wide audience and collaborate with the community, the GitHub platform is preferable. Migrating from the TASTE wiki to the GitHub pages or the project's GitHub wiki can make finding information easier.

One of the most pressing issues with the wiki at this time is that it is not up to date with the most recent version of the tool and is not updated regularly. We believe that by working together on GitHub to identify inconsistencies in the documentation, the community can play

an important role in this area. Similarly, using GitHub's bug/improvement tracking system or discussions on GitHub can contribute to collaboration on the project.

Human-Computer Interaction

The user experience of the GUI is lacking, but a new feature called SpaceCreator is in the works to improve it. There is no evidence that implementing the IDE's "Remote Dev Env" feature with the current tool is technically impossible. The WebAssembly Converter (WASM) is provided by Qt, the technology used to create the IDE, to expose a graphical interface in a Web browser. Although WebAssembly can be used to create a Qt application, it does not seem to be the best solution compared to other next-generation web framework. We recommend using more web-oriented technologies, such as those used in other open source frameworks.

Reusability

TASTE should provide an "AppStore" for space projects that offers a cross-platform/shared service. This would allow developers to quickly model their system topology, for example when evaluating a project for a tender. This AppStore would contain the components of the Kazoo demonstration project.

When starting a new project, TASTE can also prompt the user to choose between starting with a reference architecture model compliant with international standards and starting with an empty project.

This AppStore could contain components that implement protocol standards such as EDS, SPP or PUS. It would be up to TASTE, not the user, to modify these components as the standards evolve.In this regard, DataView interfaces should be able to be created using an XML schema that conforms to the SOIS EDS standard.

Supportability / Serviceability

In the Interface View, we discovered two major issues: one involving the generation of the code skeleton and the other involving the names of authorized components.

Concerning the generation of a component's code skeleton, the problem stems from the iterative redefinition of the component's ontology, i.e. the redefinition of a component's service and its interactions with the outside world. If we change the number of interfaces or the names of the interfaces after defining a component and generating the code skeleton, the header file is overwritten but not the C source file. As a consequence, we believe that following the philosophy of NASA/JPL's F Prime would be far more interesting. To put it another way, once the code skeleton is generated, all code skeleton files are generated with an extra ".tmpl" extension that the developer must remove as soon as he starts writing to the file. This would allow the developer to:

- Customize the header file to define his macros which is currently impossible because the file is overwritten at each generation
- Readjust the source file more easily to changes in the component specifications made in Interface View. Currently we have to adjust manually by copying and pasting the function declarations from the header file into the source file.

Interface View also has a problem with the user's ability to name his components. We discovered a possible conflict between the names generated by the system and those generated automatically. We discovered that there was a "Time" component that was required. The header file's name was "time.h," which conflicted with the name of the operating system's header file of the same name.

As a result, similar to ASN.1 SCC types, we believe that code skeleton generation should concatenate the auto generated names with the project name to avoid conflicts.

Installability

The method used to distribute the tool to users seemed unusual. Indeed, it seems that a Debian VM preconfigured with TASTE by the development team is required. It might be more interesting to allow the user to completely rebuild the TASTE environment on the Linux distribution they know best, such as Ubuntu or WSL-2.

All TASTE user interfaces, the IDE and GSE, could be converted into web applications, allowing containerization and the creation of a remote development environment.

9. DATA EVALUATION

Due to the lack of clear and precise requirements in the quality attributes, it was decided that the quality of each framework would be assessed at the end of the study. The definitions and semantics associated with each attribute do not constitute apodictic knowledge because they are based on scientific consensus among working groups of international experts in the field of application, as well as our empirical experiences.

Given the time constraints, this appears to be the best way to proceed with this study because it provides a number of advantages.

The most valuable advantage is that it allows for the qualitative nature of the research to be taken into account, such as the latent possibilities of each setting. Young frameworks would suffer if they were evaluated at a specific point in time without taking into account their latent potential. While a young framework may not be on par with older frameworks in some ways, it may have a stronger foundation that allows it to more easily fill the gap. In other words, while young frameworks may not meet the criteria in a factual sense, they may meet it virtually.

Evaluating these attributes is challenging; some require extremely advanced expertise, such as in security, whereas others are more about the user's use of the framework. Some frameworks would have scores in certain attribute categories, while others would not for the aforementioned reasons. If the disparity in scores persisted, the overall score for each framework would be skewed and factually inaccurate.

Finally, based on what we have seen and what we would ideally expect, we can create an informal rating scale with thresholds between each level. It is critical to remember that all three frameworks are of exceptional quality and are considered among the best in their application domain. Their editors have created high-quality work that has been painstakingly planned over a number of years in order to achieve their goal of developing a software framework as they define it.

If one framework is added to compare its quality to the other three, users must become familiar with all three frameworks before establishing their own scores; this is the method's undeniable flaw.

9.1. SCORING SYSTEM

The summary table with the scores of each attribute assigned to each framework can be found on the following page. For scoring, we use the American academic scoring system, with the addition of a N/A score for non-assessed attributes.

Grading system			
Letter Grade	**Percentage**	**Numerical value**	**Definition**
A	90-100%	4	Excellent
B	80–89%	3	Satisfactory
C	70–79%	2	Average, pass
D	60–69%	1	Needs for improvement
F	≤59%	0	Unsatisfactory
N/A	N/A	-1	Not evaluated

9.2. Framework Quality Attribute Scoring

Quality Attribute		F Prime Score	cFS Score	TASTE Score
Compatibility	Coexistence	3	3	3
	Interoperability	3	3	0
Usability	Appropriateness recognizability	2	3	1
	Learnability	2	3	1
	Popularity	4	3	1
	Operatibility	2	3	2
	Human-Computer Interaction	3	2	2
	Accessibility	3	2	4
Reliability	Maturity	4	4	1
Security	Binary-Artifacts	3	3	1
	Branch-Protection	3	4	1
	CI-Tests	4	4	2
	CII-Best Practices	3	3	1
	Code Review	4	4	4
	Contributors	3	3	2
	Dangerous Workflow	2	2	2
	Dependency-Update-Tool	0	0	0
	Fuzzing	0	4	0
	Licence	4	4	3
	Maintained	4	4	2
	Pinned-Dependencies	3	3	3
	SAST	3	4	0
	Security Policy	2	4	2
	Signed Releases	3	4	2
	Token-Permissions	2	4	1
	Vulnerabilities	3	4	3
Maintainability	Modularity	3	3	1
	Reusability	3	3	1
	Analysability	2	2	3
	Modifiability	2	4	2
	Testability	4	4	2
	Supportability	4	3	1
	Scalability	2	2	2
Portability	Adaptability	3	1	2
	Installability	4	3	2
	Replaceability	3	3	2

9.3. RADAR CHART ANALYSIS

A radar chart is used to visualize this data, making it easier to understand and learn from. This graph appears to be the best fit because it allows us to visualize multivariate data as a two-dimensional graph with many quantitative variables represented on axes that start at the same points.

9.3.1. F´ Analysis

Regarding F´, its attributes are more than satisfactory in terms of maintainability and portability, with excellent supportability. We can see that there is some progress to be made in security, such as the implementation of the Fuzzing test. We are concerned about interoperability because the method of specifying interfaces does not conform to the SEDS standard, for example.

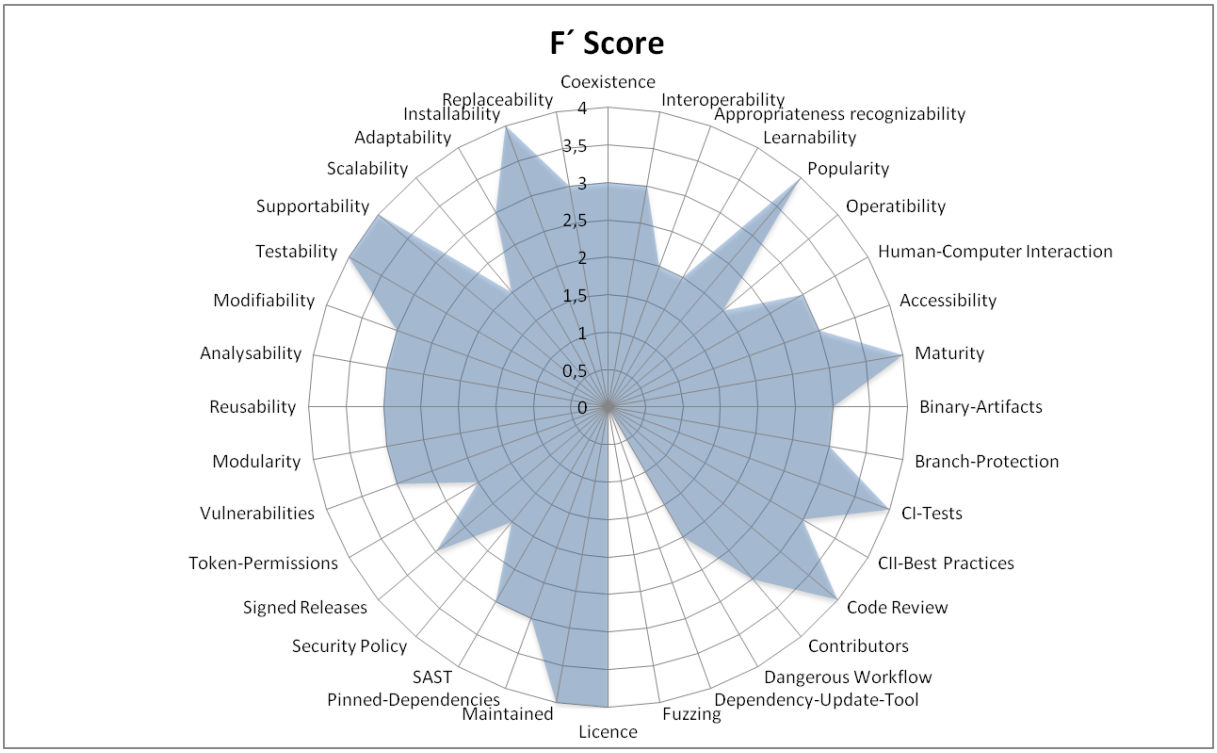


Figure 9-1: F´ Score

9.3.2. cFS Analysis

As far as cFS is concerned, its attributes are more than satisfactory in terms of compatibility and usability, with excellent security, modifiability and maturity. Its weakest points are human-computer interaction and adaptability.

For the human-computer interaction, it results from a poor user experience and the impossibility to transfer floating data.

For adaptability, it is a bit trickier because even though it supports RTEMS by default, unlike F´, cFS does not support baremetal systems by default, unlike F´. Furthermore, their adaptability scores differ because adding baremetal support to cFS appears to be more challenging than adding RTEMS support to F´.

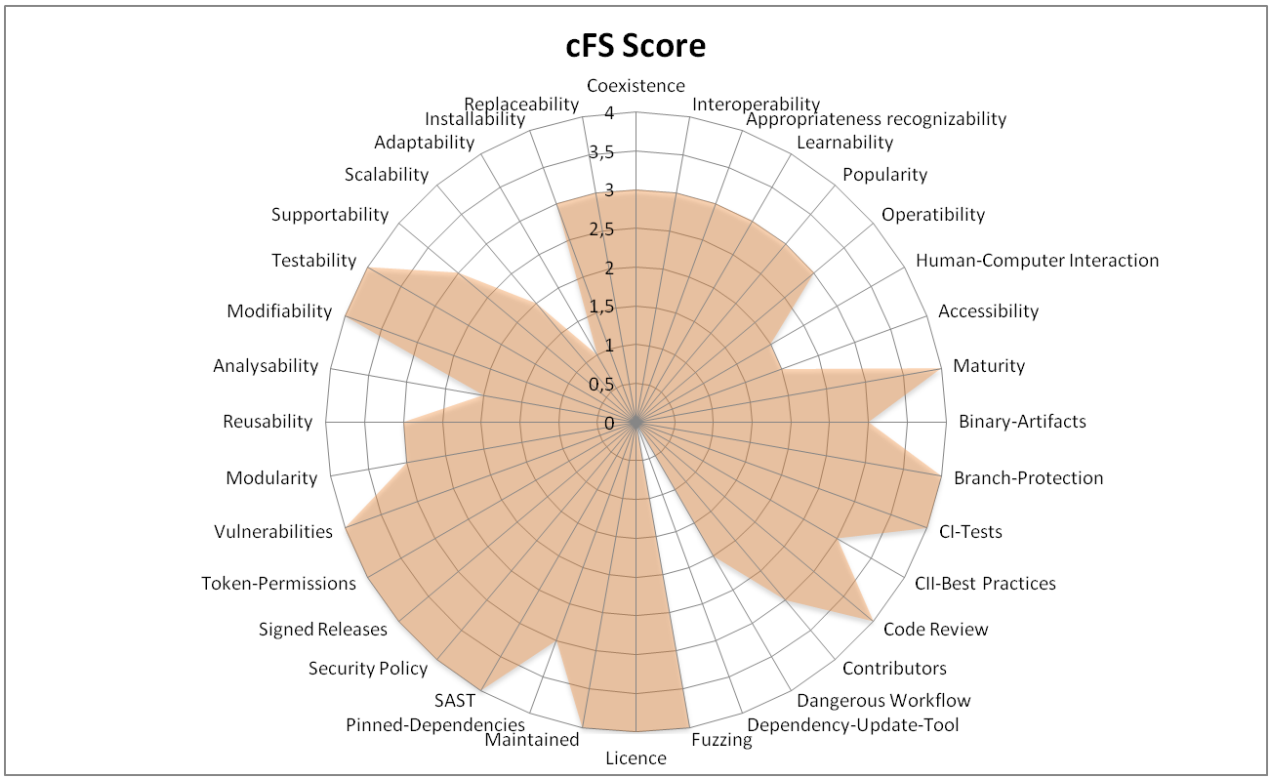


Figure 9-2: cFS Score

9.3.3. TASTE Analysis

As far as TASTE is concerned, its attributes are more than satisfactory in terms of analysability and coexistence. The analyzability score is accounted for by the MBSE approach, which allows us to graphically visualize the system's topology and share synthetic data among project stakeholders. Its coexistence score is due to its Time and Space Partitioning architecture. Its most serious flaws are interoperability, learnability and popularity.

The low interoperability score is due to the apparent difficulty in using SEDS, as well as the apparent coupling of the framework with the ASN.1 language, which appears to be greedy in memory resources. Due to a lack of tutorials and community, it is entirely possible that we lack the necessary information and expertise to verify the veracity of our claims.

This brings us back to the popularity of TASTE and the lack of an international community accessible via constantly active discussion forums.

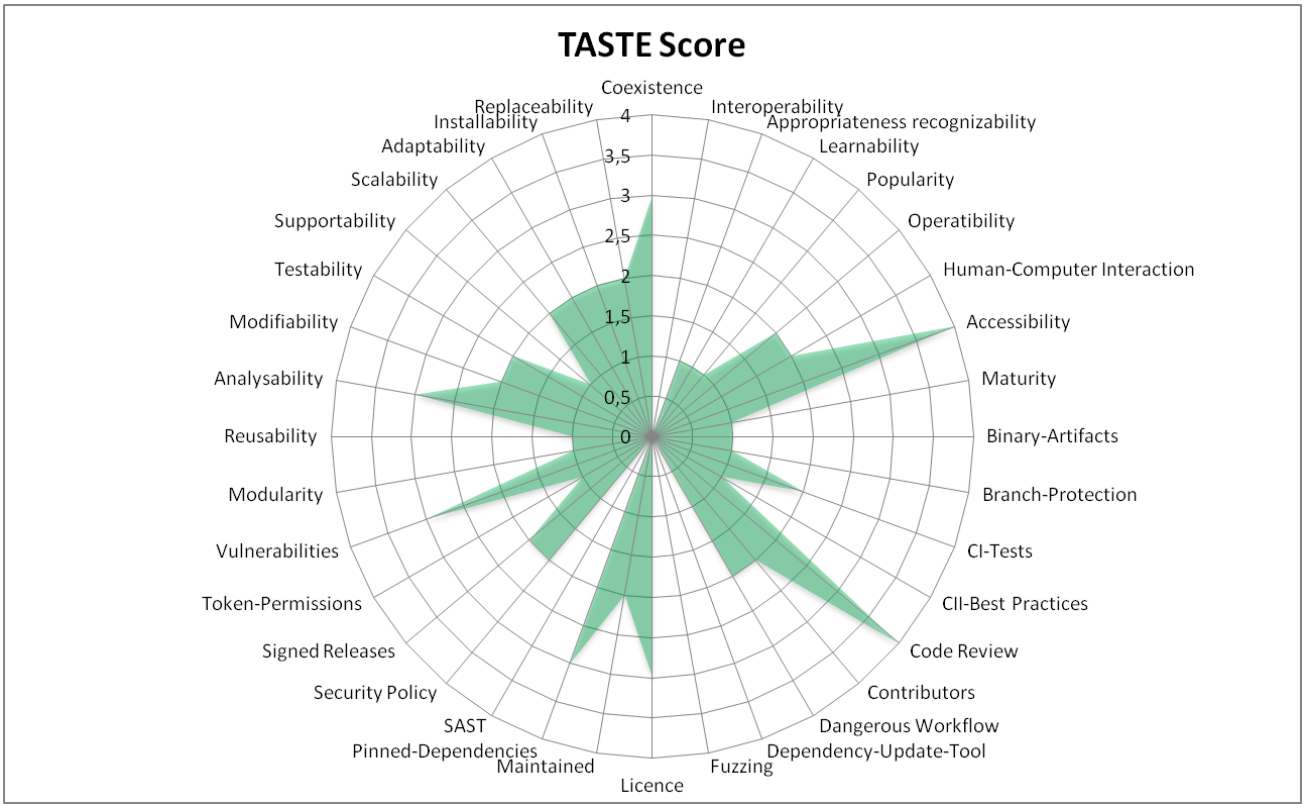


Figure 9-4: TASTE score

9.3.4. Overall Analysis

The overall radar graph shows that F´and cFS are currently outperforming TASTE. It is more difficult to determine which of F´ and cFS is superior.

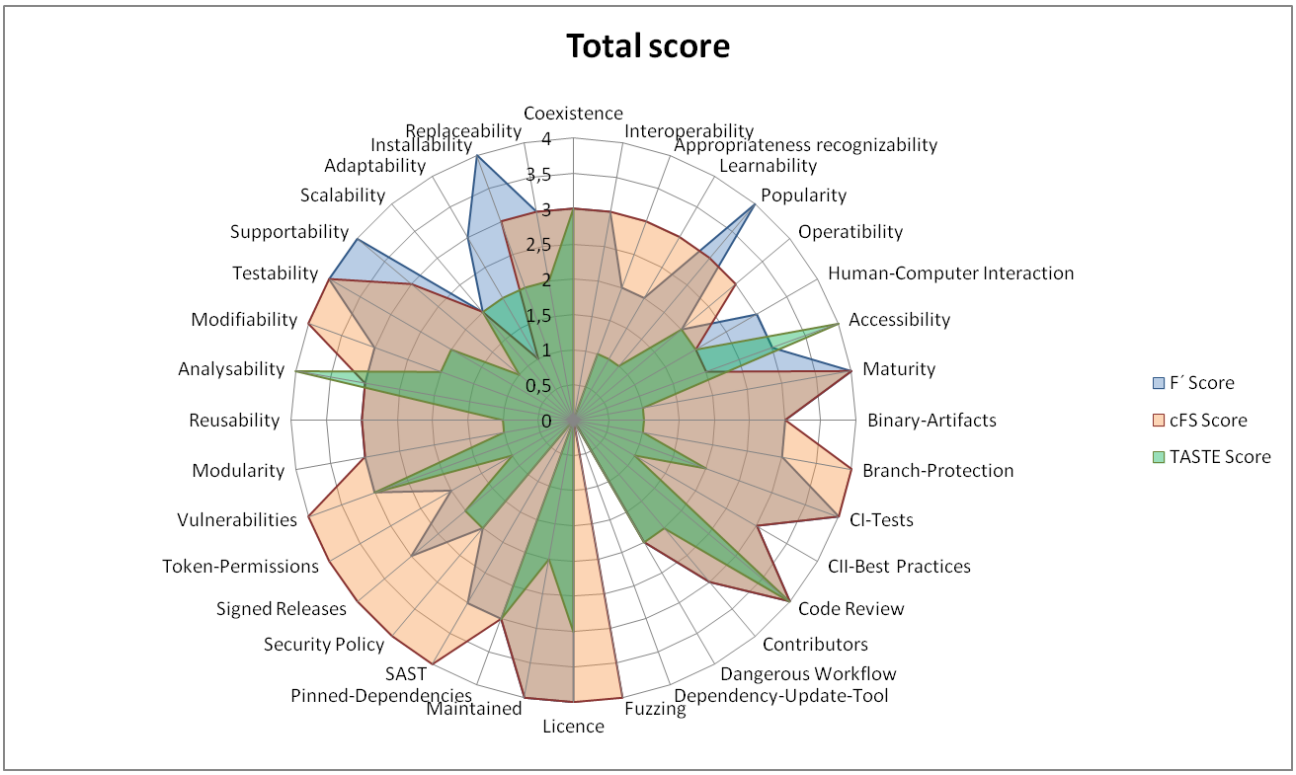


Figure 9-5: Total score

Indeed, cFS is more compatible because it integrates standard protocol implementations (CCSDS SPP and soon CCSDS SEDS). While F´ leaves the freedom to the users to implement them, which is not a major disadvantage, but it requires an effort from the user because no ready-to-use component does it.

However, three attributes affecting three key attributes cause us to clearly prefer one framework over the other. Adaptability, Human-Computer Interaction and Popularity are the three characteristics.

In terms of adaptability, unlike F´, cFS does not support baremetal, making cFS adoption difficult to implement on a program level because some projects may not have enough memory to support an OS and a file system.

Unlike F´, cFS does not support floating data exchanges due to a Ground System limitation. As a result of this flaw, the stakeholder lacks a functional, ready-to-use framework.

Unlike cFS, F´ has a support tool that allows for the auto-generation of components based on the project's design pattern, allowing for faster product development for repetitive tasks prone to human error.

Finally, while cFS is better known in the space industry than F´, F´ is better known in the open source community, according to GitHub statistics. Meanwhile, we felt that F´ operates more transparently and openly than cFS because they seem to be less constrained by government/legal/intellectual property constraints.

As a result, it appears that F´, rather than cFS, is the framework with greater operational and adaptability capabilities, as well as potential, that is better suited to our needs.

TASTE has one of the most advanced visions of software development with its MBSE approach. We can abstract from any programming language by macroscopically specifying the topology of the software system in a graphical modeling language. We can specify that a component will be written in the language best suited to its functionality. If the Rust language becomes dominant in the space sector in the future, we assume that adding Rust support to the framework will be relatively straightforward.

10. CONCLUSION AND FUTURE WORK

Recommended Framework

We have seen that the frameworks under consideration have slightly different objectives with varying operational requirements.

cFS is designed more for large satellites that require on-board maintenance, extensive data storage, non-communication operations/recording, and table/configuration management. This is demonstrated by the fact that the framework requires a file system to facilitate these activities.

F´ is better suited to small satellites or equipment with a more static configuration.

TASTE is applicable to any type of system because it does not provide a reference architecture. The user is free to design his own architecture according to his needs.

If we were to make a suggestion, it would be from the perspective of a small equipment supplier. This SME would have the following characteristics:

- small development teams (from one to 4 people);
- small project/program budgets (staff hours ranging from hundreds for assessments to tens of thousands for large projects);
- small budgets for continuous improvement and R&D (staff hours often do not exceed one thousand hours per year);
- strong competition resulting in ever shorter time-to-market;
- the need to meet standards compliance requirements such as SEDS or reference architectures (CCSDS, SAVOIR-OSRA).

To reduce recurring costs and time-to-market, this SME requires a software framework that is as versatile as possible for use in any operational context and, therefore, with variable availability of hardware resources (volatile and non-volatile memory ranging from MB to several hundred MB).

In this context, we believe that the F´ framework currently best meets the needs of this stakeholder. We are confident in our diagnosis because F´ editors claim that it can be used on instruments. However, when choosing a framework, it is essential to favour frameworks whose development teams are frequently confronted with the same problems as the user. In our case, these are spaceflight software systems embedded in instruments and payloads with limited hardware resources, which may preclude the use of an operating system or file system.

However, as past performance is not indicative of future results, technical watch of the entire open source framework ecosystem must be done on an ongoing basis. Other frameworks, such as Blue Origin's SpaceROS or CAST's FUHSI, would be worth investigating once they become open source.

Concerns about Framework Governance

Governance is an important aspect of Open Source frameworks. A framework is created by an official team of maintainers who are part of the organization that dictates its direction. The community must express its most pressing needs in order for the framework to evolve and become more community-oriented. Indeed, even if the application field is common, the maintenance team is confronted with its own problems or those of its closest collaborators. They are unable to guess or anticipate the specific problems and needs of all stakeholders.

It is then each stakeholder's responsibility to express their needs in order to steer the framework roadmap in an effective direction for long-term use.

Without this feedback, the framework's evolutions may coincidentally meet the needs of the maintenance team and a stakeholder's team. Otherwise, a stakeholder will leave the community in search

of alternative frameworks, or even incur the cost of developing its own framework to meet only its short- or medium-term operational needs.

One of our concerns is the organization that governs the geopolitical dimension of a framework. Indeed, if an actor from one country chooses to use a framework from another country, the digital sovereignty of this country is questioned. Consequently, this country may wish to avoid adopting this framework and create its own.

We believe that it would be interesting to be inspired by the RISC-V neutral governance model. This Open Standard Instruction Set Architecture (ISA) is managed by an international non-profit organization, so it is not affected by geopolitical issues. The organization that manages the framework would resemble various stakeholders to achieve the broadest and most effective open collaboration possible. This organization would provide leadership to the technical endeavor by defining the long-term roadmap for the framework.

Third party ground system

One of the recurring flaws we have encountered in frameworks is the lack of a way to visualize data on the control screen. Of course, the default GUI is suitable for rapid prototyping and interacting with embedded software systems via simple commands and telemetry messages. However, when attempting to manipulate more complex message definitions such as structures, images, video streams, or even simple floating numbers, we quickly run into limitations. When faced with this problem, the development team has two options.

Either adapt or modify the source code of the GUI to gradually remove these constraints, or use third-party Mission Control Software with far more advanced data visualization features. This second option appears to us to be the most appealing. A third option is available if the user already has a similar graphical interface that can be linked to the embedded software system. However, we begin with the assumption that a framework must provide all of the elements required for a startup to focus on its mission, and that the startup has no prior experience developing such a GUI.

The disparity in the size of stakeholder development teams is noteworthy. Due to a lack of human and financial resources, the flight software system's developers must modify the GUI for some stakeholders, SMEs, or startups. These HMIs, on the other hand, are frequently implemented in programming languages such as Javascript, in which the development team has little or no expertise due to the languages' being outside of their domain of expertise.

Given the abundance of open source mission control software available, as well as the aforementioned issue, it appears that a standardized representation of the message formats exchanged between the two systems is the most effective solution.

The CCSDS SOIS EDS standard is currently the most likely to be adopted by the space community. We will be able to more easily integrate mission control software into an open source flight software framework when each stakeholder complies with this standard.

Looking to the future

> “The future is already here - it's just not very evenly distributed.”
> – William Gibson

We are on the verge of a significant shift in the way software in the space sector is developed. The industry's adoption of open source frameworks for the development of commercial components exemplifies this transformation.

In terms of these frameworks' long-term viability and support, we believe that stakeholders who have built their businesses on them should contribute to their expansion. This can be accomplished financially by paying the framework publishers a portion of the profits. Direct contributions to the source code can also be made by providing feedback on internal project improvements or by mobilizing agents on the framework itself.

Stakeholders who join and thrive in this open source environment will gain a significant competitive advantage through greater technical responsiveness and less technical debt.

11. BIBLIOGRAPHY

In this section we list all the articles that served as sources for the statements made in this paper. We also invite the reader to go back to the source of the statements made in the document read from the bibliographic references to avoid misinterpretation. The following list, in IEEE citation format, specifies the reference documents that have been considered during the study. Reference documents follow are referred to by [n] where n is an incremental integer.

[1] "CCSDS.org - The Consultative Committee for Space Data Systems (CCSDS)." https://public.ccsds.org/default.aspx

[2] B. F. Collins, "Development of a space universal modular architecture (SUMO)," in *2013 IEEE Aerospace Conference*, Mar. 2013, pp. 1–9. doi: 10.1109/AERO.2013.6497429.

[3] "Future Airborne Capability Environment (FACE™)," *The Open Group Website*, Jun. 02, 2020. https://www.opengroup.org/face.

[4] "SAVOIR - Space AVionics Open Interface aRchitecture." https://savoir.estec.esa.int/

[5] "Building Evolutionary Architecture." https://www.thoughtworks.com/books/building-evolutionary-architectures.

[6] "Robert C. Martin," *Wikipedia*. Sep. 29, 2021. Available: https://en.wikipedia.org/w/index.php?title=Robert_C._Martin&oldid=1047162253

[7] R. C. Martin, "Amazon.fr - Clean Architecture: A Craftsman's Guide to Software Structure and Design: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series) - Martin, Robert C. - Livres." https://www.amazon.fr/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164

[8] "Who Uses SpaceWire?," *STAR-Dundee*. https://www.star-dundee.com/spacewire/getting-started/who-uses-spacewire/

[9] "Time-Triggered Ethernet Slims Down Critical Data Systems | NASA Spinoff." https://spinoff.nasa.gov/Spinoff2018/t_4.html

[10] "CCSDS - SPACECRAFT ONBOARD INTERFACE SERVICES— XML SPECIFICATION FOR ELECTRONIC DATA SHEETS." Available: https://public.ccsds.org/Pubs/876x0b1.pdf

[11] B. Ascension 2022, "Mission Ready Software," *Bright Ascension*. https://brightascension.com/.

[12] "On board satellite software." https://www.terma.com/markets/space/space-segment/satellite-data-handling/.

[13] D. Miranda, M. Ferreira, F. Kucinskis, and D. Mccomas, "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions," *Journal of Aerospace Technology and Management*, vol. 11, Oct. 2019, doi: 10.5028/jatm.v11.1081.

[14] CCSDS, "CCSDS - CAST FLIGHT SOFTWARE AS A CCSDS ONBOARD REFERENCE ARCHITECTURE." Available: https://public.ccsds.org/Pubs/811.1o1e1.pdf

[15] J. Baragry and K. Reed, "Why is it so hard to define software architecture?," Jan. 1999, pp. 28–36. doi: 10.1109/APSEC.1998.733577.

[16] "Ralph Johnson - Biography," *IEEEXplore*. https://ieeexplore.ieee.org/author/37306473000

[17] "Software Architecture Guide," *martinfowler.com*. https://martinfowler.com/architecture/

[18] "Martin Fowler Biography," *martinfowler.com*. https://martinfowler.com/.

[19] "Is High Quality Software Worth the Cost?," *martinfowler.com*. https://martinfowler.com/articles/is-quality-worth-cost.html.

[20] R. C. Martin, “Clean Architecture - Clean Coder Blog.” https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[21] “Alistair Cockburn - Hexagonal Architeture,” *Alistair Cockburn*. https://alistair.cockburn.us/.
[22] “Alistair Cockburn,” *Wikipédia*. Apr. 16, 2021. Available: https://fr.wikipedia.org/w/index.php?title=Alistair_Cockburn&oldid=181968563
[23] “The Onion Architecture : part 1 | Programming with Palermo.” https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
[24] “Jeffrey Palermo.” https://mvp.microsoft.com/fr-fr/PublicProfile/33697?fullName=Jeffrey%20Palermo.
[25] “SOLID,” *Wikipedia*. Dec. 21, 2021. Available: https://en.wikipedia.org/w/index.php?title=SOLID&oldid=1061447159
[26] “Domain-driven design,” *Wikipedia*. Nov. 22, 2021. Available: https://en.wikipedia.org/w/index.php?title=Domain-driven_design&oldid=1056545288
[27] “Eric Evans,” *Domain Driven Design Europe*. https://2018.dddeurope.com/speakers/eric-evans.
[28] “NASA Exoplanet Exploration - Star Tracker,” *Exoplanet Exploration: Planets Beyond our Solar System*. https://exoplanets.nasa.gov/star-tracker.
[29] M. McCrum and P. Mendham, “SmallSat Conference - Integrating Advanced Payload Data Processing in a Demanding CubeSat Mission,” *Small Satellite Conference*, Aug. 2015. Available: https://digitalcommons.usu.edu/smallsat/2015/all2015/140
[30] “CCSDS - SPACECRAFT ONBOARD INTERFACE SERVICES.” Available: https://public.ccsds.org/Pubs/850x0g2.pdf
[31] “Martin Fowler - Microservices,” *martinfowler.com*. https://martinfowler.com/articles/microservices.html.
[32] “CCSDS - MISSION OPERATIONS SERVICES CONCEPT.” Available: https://public.ccsds.org/Pubs/520x0g3.pdf
[33] John, “LEGO blocks and organ transplants,” Feb. 03, 2011. https://www.johndcook.com/blog/2011/02/03/lego-blocks-and-organ-transplants/.
[34] E. Wyrwas and C. Szabo, “Proton Testing of AMD Ryzen 3 1200 Microprocessors,” Art. no. 2019-561-NEPP, Jul. 2019. Available: https://ntrs.nasa.gov/citations/20190031853
[35] “Abstraction Layer | Sweetwater,” *inSync*, Jan. 17, 2017. https://www.sweetwater.com/insync/abstraction-layer/.
[36] “Ew Dijkstra Quotes.” https://wiki.c2.com/?EwDijkstraQuotes.
[37] “Operating system abstraction layer,” *Wikipedia*. Apr. 25, 2020. Available: https://en.wikipedia.org/w/index.php?title=Operating_system_abstraction_layer&oldid=953026304
[38] “What is a Hardware Abstraction Layer (HAL)? - Definition from Techopedia,” *Techopedia.com*. http://www.techopedia.com/definition/4288/hardware-abstraction-layer-hal
[39] *GitHub SAMRH71 BSP - tc interrupt*. Microchip MPLAB Harmony, 2021. Available: https://github.com/Microchip-MPLAB-Harmony/csp_apps_sam_rh71/blob/master/apps/tc/tc_compare_mode/firmware/src/config/sam_rh71_ek/interrupts.c
[40] *GitHub SAMRH71 BSP - PioLed startup file*. Microchip MPLAB Harmony, 2021. Available: https://github.com/Microchip-MPLAB-Harmony/csp_apps_sam_rh71/blob/master/apps/pio/pio_led_on_off_interrupt/firmware/src/config/sam_rh71_ek/startup_xc32.c

[41] A. Ostrowski and P. Gaczkowski, *Software Architecture with C++: Design modern systems using effective architecture concepts, design patterns, and techniques with C++20*, 4e édition. Birmingham Mumbai: Packt Publishing, 2021.
[42] "CCSDS - SPACE PACKET PROTOCOL." Available: https://public.ccsds.org/Pubs/133x0b2e1.pdf
[43] "NASA - cFS Overview." Available: https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf
[44] E. J. Timmons, "Core Flight System (cFS) Training - cFS Caelum," NASA/TM–20205000691/REV 2, Oct. 2021. Available: https://ntrs.nasa.gov/citations/20210022378
[45] Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani, "Software Architecture: The Hard Parts." https://www.oreilly.com/library/view/software-architecture-the/9781492086888/.
[46] M. Fowler, "Microservices and SOA," *martinfowler.com*. https://martinfowler.com/articles/microservices.html.
[47] "ISO 25010." https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.
[48] L. Lazic, A. Kolašinac, and D. Avdić, "The Software Quality Economics Model for Software Project Optimization," *WSEAS Transactions on Computers*, vol. 8, Jan. 2009.
[49] "Quality Management - SEBoK." https://www.sebokwiki.org/wiki/Quality_Management#:~:text=Quality%20attributes%2C
[50] "A Quality Attributes Guide for Space Flight Software Architects." https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=454600.
[51] J. Wilmot, L. Fesq, and D. Dvorak, "NASA Technical Reports Server - Quality Attributes for Mission Flight Software: A Reference for Architects," presented at the IEEE Aerospace Conference, Big Sky, MT, Mar. 2016. doi: 10.1109/AERO.2016.7500850.
[52] "Evaluation of Software Product Functional Suitability: A Case Study." Available: https://www.aqclab.es/images/AQCLab/Noticias/SQP/software-quality-management-evaluation-of-software-product-functional-suitability-a-case-study.pdf
[53] "Hardware Engineering Design - System Coexistence." https://www.hwe.design/system-testing/system-coexistence.
[54] "Operability," *Wikipedia*. Dec. 07, 2020. Available: https://en.wikipedia.org/w/index.php?title=Operability&oldid=992838809
[55] "RDI: The Future of Remote Development," *Border*, Jan. 20, 2020. https://borderux.com/remote-development-is-the-future/ .
[56] I. Tzinis, "NASA - Technology Readiness Level," *NASA*, May 06, 2015. http://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level.
[57] "NASA - Fault-Detection, Fault-Isolation and Recovery (FDIR) Techniques." https://llis.nasa.gov/lesson/839.
[58] "Open Source Security Foundation (OpenSSF)," *GitHub*. https://github.com/ossf/.
[59] *Security Scorecards*. Open Source Security Foundation (OpenSSF), 2022. Available: https://github.com/ossf/scorecard
[60] *Security Scorecards - Check Documentation*. Open Source Security Foundation (OpenSSF), 2022. Available: https://github.com/ossf/scorecard/blob/ab16cdbbc20fa2c2d53c9c02f382a95d27d342d0/docs/checks.md
[61] *King of bugs found by Fuzzing*. Google, 2022. Available: https://github.com/google/fuzzing/blob/748de3eed506a515840fcea85fb1c0c22c170c2a/docs/why-fuzz.md

[62] “Static Code Analysis Control | OWASP Foundation.” https://owasp.org/www-community/controls/Static_Code_Analysis.
[63] “Quality Assurance - Software Quality Attributes.” https://www.qasigma.com/2008/12/software-quality-attributes.html#:~:text=Portability%3A
[64] H. Subramaniam and H. Zulzalil, “Software Quality Assessment using Flexibility: A Systematic Literature Review,” *International Review on Computers and Software*, vol. 7, pp. 2095–2099, Sep. 2012.
[65] “Logiciels embarqués spatiaux - LESIA - Observatoire de Paris.” https://lesia.obspm.fr/-Logiciels-embarques-.html?artpage=4.
[66] “QEMU.” https://www.qemu.org/.
[67] “Renode.” https://renode.io/.
[68] “Gitpod - Always ready to code.” https://www.gitpod.io/.
[69] “GitHub Codespaces,” *GitHub*. https://github.com/features/codespaces.
[70] “VSCode Dev Container.” https://code.visualstudio.com/docs/remote/containers
[71] B. W. Boehm, “NASA - SWE-019 - Software Life Cycle,” *Computer*, vol. 21, no. 5, pp. 61–72, May 1988, doi: 10.1109/2.59.
[72] “10 Types of Data Visualization Made Simple (Graphs & Charts),” *Boost Labs*, Sep. 14, 2019. https://boostlabs.com/blog/10-types-of-data-visualization-tools/.
[73] “Web application architecture: Components, models and types.” https://www.scnsoft.com/blog/web-application-architecture
[74] D. C. Contributor, “Google’s ‘20% rule’ shows exactly how much time you should spend learning new skills—and why it works,” *CNBC*, Dec. 16, 2021. https://www.cnbc.com/2021/12/16/google-20-percent-rule-shows-exactly-how-much-time-you-should-spend-learning-new-skills.html.
[75] “Dev corrupts NPM libs ‘colors’ and ‘faker’ breaking thousands of apps,” *BleepingComputer*. https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps/.
[76] “Log4Shell,” *Wikipédia*. Feb. 01, 2022. Available: https://fr.wikipedia.org/w/index.php?title=Log4Shell&oldid=190427998
[77] A. M. St Laurent, “Understanding Open Source and Free Software Licensing,” *OReilly*. https://www.oreilly.com/library/view/understanding-open-source/0596005814/
[78] European Commission, “European Commission - Open source software strategy,” *European Commission - European Commission*. https://ec.europa.eu/info/departments/informatics/open-source-software-strategy_en.
[79] J. Castex, “Légifrance - Droit national en vigueur - Circulaires et instructions - Circulaire n°6264/SG du 27 avril 2021 relative à la politique publique de la donnée, des algorithmes et des codes sources.” https://www.legifrance.gouv.fr/download/pdf/circ?id=45162.
[80] “Spacearth Initiative | Dossiers.” https://www.spacearth-initiative.fr/.
[81] “Spacearth Initiative | Nos propositions.” https://www.spacearth-initiative.fr/nos-propositions/.
[82] “How to navigate open source licensing risks,” *Trend Micro*, Jul. 08, 2021. https://www.trendmicro.com/en_us/research/21/g/navigating-open-source-licensing-risk.html.
[83] “The Innovator’s Dilemma: When New Technologies Cause Great Firms to Fail,” *Clayton Christensen*. //claytonchristensen.com/books/the-innovators-dilemma/.
[84] “Rough Order of Magnitude (ROM) Costs and Detailed Cost Estimates – Communications Service Office.” https://cso.nasa.gov/content/roms/.

[85] E. Ries, “The Lean Startup,” *Wikipedia*. Sep. 29, 2021. Available: https://en.wikipedia.org/w/index.php?title=The_Lean_Startup&oldid=1047262199
[86] J. M. Bohlen and G. M. Beal, “The diffusion process,” *Iowa State University - Special Report*. Available: https://dr.lib.iastate.edu/entities/publication/692fb2e6-9d7b-4679-9e84-5f3985af199c
[87] Lynn, “What is the Technology Adoption Curve, and How Should You Use It?,” *Consultport*, Oct. 15, 2021. https://www.consultport.com/consulting-academy/what-is-the-technology-adoption-curve-and-how-should-you-use-it/.
[88] NASA JPL, “Meet the Open-Source Software Powering NASA’s Ingenuity Mars Helicopter,” *NASA Jet Propulsion Laboratory (JPL)*. https://www.jpl.nasa.gov/news/meet-the-open-source-software-powering-nasas-ingenuity-mars-helicopter.
[89] mars.nasa.gov, “NASA JPL - Mars Helicopter.” https://mars.nasa.gov/technology/helicopter/
[90] “F´ on Constrained Memory Devices · LeStarch/fprime Wiki · GitHub.” https://github.com/LeStarch/fprime/wiki/F%C2%B4-on-Constrained-Memory-Devices.
[91] N. Cheek, N. Daniel, E. Lightsey, S. Peet, C. Smith, and D. Cavender, “Development of a COTS-Based Propulsion System Controller for NASA’s Lunar Flashlight CubeSat Mission,” *Small Satellite Conference*, Aug. 2021. Available: https://digitalcommons.usu.edu/smallsat/2021/all2021/196
[92] J. W. Bradbury, “Open Source Core Flight System (cFS) Flight Software (FSW) Verification & Validation (V&V) Final Summary: IV&V Analysis Technical Report,” NASA/CR-20205010026, Dec. 2020. Available: https://ntrs.nasa.gov/citations/20205010026
[93] K. Hille, “Core Flight Software Chosen for Lunar Gateway,” *NASA*, Feb. 11, 2021. http://www.nasa.gov/feature/goddard/2021/core-flight-software-chosen-for-lunar-gateway.
[94] *cFE Application Developer’s Guide*. NASA, 2021. Available: https://github.com/nasa/cFE/blob/c161cf04fe7c5fd2f2a533ac1d7522b31191eab6/docs/cFE%20Application%20Developers%20Guide.md
[95] *cFE - Directory Tree*. NASA, 2021. Available: https://github.com/nasa/cFE/blob/98f78e8604c19415fd1e199eae94196a781539b8/docs/cFE%20Application%20Developers%20Guide.md
[96] *cFS - ut_assert README*. NASA, 2022. Available: https://github.com/nasa/osal/blob/4cc6dbb5019d0589d5ce52e3755a0b7a012ade3c/ut_assert/README.txt
[97] *cFS Test Framework - GitHub*. NASA, 2022. Available: https://github.com/nasa/CTF
[98] J. Hickey, *cFE EDS Framework*. 2021. Available: https://github.com/jphickey/cfe-eds-framework
[99] *NASA - EdsLib*. NASA, 2021. Available: https://github.com/nasa/EdsLib
[100] L. E. Prokop and J. J. Wilmot, “NASA’s Core Flight Software - A Reusable Real-Time Framework,” Dec. 03, 2014. Available: https://ntrs.nasa.gov/citations/20140017040
[101] “TASTE Tools Website.” https://taste.tools/
[102] “Overview - TASTE.” https://taste.tuxfamily.org/wiki/index.php?title=Overview.
[103] NASA, “NASA Software Engineering Requirements - NPR 7150.2C.” https://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7150&s=2C
[104] “Intelligent on-board flight software | CGI UK.” https://www.cgi.com/uk/en-gb/brochure/space/intelligent-onboard-flight-software.

[105] J.-L. TERRAILLON, “SAVOIR: Reusing specifications to improve the way we deliver avionics,” Toulouse, France, Feb. 2012. Available: https://hal.archives-ouvertes.fr/hal-02263427
[106] M. McCrum and P. Mendham, “Integrating Advanced Payload Data Processing in a Demanding CubeSat Mission,” *Small Satellite Conference*, Aug. 2015. Available: https://digitalcommons.usu.edu/smallsat/2015/all2015/140
[107] SAVOIR, “OSRA - Onboard Software Reference Architecture,” *ESA ESSR*. https://essr.esa.int/project/osra-onboard-software-reference-architecture.
[108] “Embedded Hypervisor | Ultimate Guides | BlackBerry QNX.” https://blackberry.qnx.com/en/ultimate-guides/embedded-hypervisor.
[109] blogs.blackberry.com, “Extend the Lifecycle of Embedded Systems with a Hypervisor.” https://blogs.blackberry.com/en/2020/02/extend-the-lifecycle-of-embedded-systems-with-a-hypervisor.
[110] J. Rexroat, “PROPOSED MIDDLEWARE SOLUTION FOR RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED NETWORKS,” *Theses and Dissertations--Electrical and Computer Engineering*, Jan. 2014. Available: https://uknowledge.uky.edu/ece_etds/63
[111] D. McComas, “OpenSatKit.” https://github.com/OpenSatKit/OpenSatKit
[112] *Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS)*. NASA, 2022. Available: https://github.com/nasa/icarous
[113] NASA, *NASA Operational Simulator for Small Satellites - NOS3*. NASA, 2021. Available: https://github.com/nasa/nos3
[114] NASA, *Open MCT*. NASA, 2022. Available: https://github.com/nasa/openmct
[115] yamcs, *Yamcs Mission Control*. Yamcs Mission Control, 2022. Available: https://github.com/yamcs/yamcs
[116] “Space Assigned Numbers Authority (SANA).” https://sanaregistry.org/r/sois/
[117] “CCSDS EDS Reference Tooling,” *ESA ESSR*. https://essr.esa.int/project/ccsds-eds-reference-tooling.
[118] “ESSR - License European Space Agency Public License (ESA-PL) Commentary – v2.3.” https://essr.esa.int/license/european-space-agency-public-license-esa-pl-commentary-v2-3
[119] “SAVOIR-EDS SECT - Repository.” Available: https://gitrepos.estec.esa.int/SAVOIR-EDS/SECT.
[120] “Vagrant by HashiCorp,” *Vagrant by HashiCorp*. https://www.vagrantup.com/

12. ANNEXE A – CHANGE HISTORY

This section lists the changes made to the document for all revisions. Changes are listed by revision, section and type. The type of change is identified in the third column of the table according to the following convention:

- "**E**" : Editorial or stylistic change;
- "**L**" : Clarification of existing text;
- "**D**" : A feature from the previous revision has been removed;
- "**C**" : A feature from the previous revision has been changed;
- "**N**" : A new feature has been introduced.

CHANGE RECORD			
Revision	**Section**	**Type**	**Description**
A	-	-	First issue

13. ANNEXE B – DEFINITIONS, ACRONYMS AND ABBREVIATIONS

13.1. SYMBOLS AND ABBREVIATIONS

The acronyms used in this document and requiring definition are included in the following table:

A

AADL	: Architecture Analysis and Design Language
AD	: Applicable Document
AFL	: American Fuzzing Loop
AIT	: Assembly, Integration and Test
AOCS	: Attitude and Orbit Control System
APU	: Application Processing Unit
ARINC	: Aeronautical Radio, INCorporated
ARM	: Advanced Risk Machine
ASN	: Abstract Syntax Notation

B

BLOB	: Binary Large Object
BSON	: Binary JSON
BSP	: Board Support Package

C

CAST	: China Academy of Space Technology
CBSE	: Component Based Software Engineering
CCSDS	: Consultative Committee for Space Data Systems
CFDP	: CCSDS File Delivery Protocol
cFE	: core Flight Executive
cFS	: core Flight System
CMD	: Command
COrDET	: Component Oriented DEvelopment Techniques
COTS	: Commercial Off-The-Shelf
CPU	: Central Processing Unit
CTF	: cFS Test Framework

D

DB	: DataBase
DDD	: Domain-Driven Design
DLR	: German Aerospace Center

E

ECSS	: European Cooperation for Space Standardization
EDS	: Electronic Data Sheet
EEPROM	: Electrically-Erasable Programmable Read-Only Memory
ESA	: European Space Agency
ESTEC	: European Space Research and Technology Centre

F

FACE	: Future Airborne Capability Environment

N

NASA	: National Aeronautics and Space Administration

O

OH	: Optical Head
OS	: Operating System
OSAL	: Operating System Abstraction Layer
OSI	: Open Systems Interconnection
OSRA	: On board Software Reference Architecture
OSS	: Open-Source Software
ORK	: Open Ravenscar Kernel

P

PE	: Processing Element
PoC	: Proof of Concept
POSIX	: Portable Operating System Interface
PSP	: Platform Support Package / Parker Solar Probe
PUS	: Packet Utilization Standard

R

RAM	: Random Access Memory
RODOS	: Realtime Onboard Dependable Operating System
RoI	: Return on Investments
ROM	: Read-Only Memory
RPU	: Real-Time Processing Unit
RTEMS	: Real-Time Executive for Multiprocessor Systems
RTOS	: Real-Time Operating System
R&D	: Research and Development

S

SANA	: Space Assigned Numbers Authority
SAVOIR	: Space AVionics Open Interface aRchitecture
SAVOIR-FAIRE	: SAVOIR – Fair Architecture and Interface Reference Elaboration
SAVOIR-SAFI	: SAVOIR – Sensor and Actuators Functional Interface
SCC	: Space Certifiable Compiler
SDK	: Software Development Kit
SDL	: Specification and Description Language
SDU	: Service Data Unit
SEDS	: SOIS EDS
SM	: Software eleMent
SME	: Small and Medium-sized Enterprise
SOA	: Service-Oriented Architecture
SoC	: System on a chip

FOSS : Free and Open Source Software

FSW : Flight SoftWare

G

GDS : Ground System
GERICOS : GEneRIC Onboard Software
GNC : Guidance, Navigation, and Control
GSE : Ground Support Equipment
GSFC : Goddard Space Flight Center
GUI : Graphical User Interface

H

HAL : Hardware Abstraction Layer

I

ICD : Interface Control Document
IEC : International Electrotechnical Commission
IEEE : Institute of Electrical and Electronics Engineers
IPC : Inter Process/Partition Communication
ITT : Invitation to Tender
ISO : International Organization for Standardization

J

JPL : Jet Propulsion Laboratory
JSC : Johnson Space Center
JSON : JavaScript Object Notation

K

KB : KiloByte
KDP : Key Decision Point

M

MB : Mega Byte
MBSE : Model Based Software Engineering
MCT : Mission Control Technologies
MIT : Massachusetts Institute of Technology
MO : Mission Operation
MOSS : Mission Operations Services Standard
MP : MultiProcessing
MSC : Message Sequence Charts
MSFC : Marshall Space Flight Center
SODERN : SOciété D'Etudes et Réalisations Nucléaires (Company of nuclear studies and achievements)
SOIS : Spacecraft Onboard Interface Service
SPA : Space Plug-and-play Architecture
SPP : Space Packet Protocol
SW : SoftWare

T

TAI : Temps Atomique International - International Atomic Time
TASTE : The ASSERT Set of Tools for Engineering
TC : TeleCommand
TDD : Test Driven Development
TLM : Telemetry
TM : TeleMetry
TRL : Technology Readiness Level
TSP : Time and Space Partitioning
TT : Time-Triggered

U

UTC : Universal Time Coordinated

V

VHDL : VHSIC Hardware Description Language
VHSIC : Very High Speed Integrated Circuit

X

XML : Extensible Markup Language
XTCE : XML Telemetric and Command Exchange
xTEDS : eXtensible Transducer Electronic Data Sheet

Y

YAMCS : Yet Another Mission Control System

13.2. DEFINITIONS

For the purpose of this study report, the following definitions apply. Some of these definitions can be found in DO-297, CCSDS, ECSS, and SAVOIR.

Term	Definition
BLOB	Structure used to store the binary attributes of objects. It is a variable-length, unbounded, byte array. This type can allow language mappings and encodings to use more efficient or appropriate representations.
Building block	A component that: • Has a clear, open, well-defined, specified and documented function and interfaces. • Is worth developing, i.e., its use is envisioned for the bulk of space missions • meets defined performance, operational and other requirements • Is self-contained in such a way that it is compatible with use at a higher level of integration, e.g. board, equipment, subsystem. • The composability and compositionality of its properties are guaranteed • has an accessible TRL and quality level • Is applicable in a well-defined physical and software environment • Is derived from a process that can be repeated with guarantees • Is designed to be reused by different users, in different projects (it can be configured according to variability factors) • Can be made available off-the-shelf, under defined conditions.
Component	A logical element of a system accessed through defined interfaces. A component may be purely conceptual or realized in software or hardware. It does not provide an aircraft function by itself.
Core SW	OS and support SW that manage resources to provide an environment in which application can execute. Core software is a necessary component of a platform and is typically comprised of one or more modules.
Device	A physical element of a system accessed through subnetworks-layer interfaces
Interface	A facility provided or supplied by a component that allows exchange of data
Module	A component of collection of components that may be accepted by themselves. A module also comprises other modules. A module may be software, hardware or a combination of hardware and software, which provided resources the system.
Open Source Software	Software whose human-readable source code is made freely available to the public under an OSS license, which provides for terms of use, reuse, modification/improvement and redistribution. Often, the development, management, and planning of the software is done publicly or is readily observable by an individual or organization not previously connected with the OSS.
Partitioning	An architectural technique to provide the necessary separation and independence of function or application to ensure that only intended coupling occurs. The mechanisms for providing the protection in a platform are specified to a required level of integrity.
Platform/ System Executive Platform (SEP)	Module or modules including core SW that manages resources to support at least one application. The platform established a computing environment, support services, and platform-related capabilities, such as health monitoring and fault management.
Resource	Any object (process, memory, software, data, etc.) or component used by application. A resource may be shared by multiple applications or dedicated to a specific.
Service	A set of capabilities that a component provides to another component via an interface. A service is defined in terms of a set of operations that can be invoked and executed via the service interface.

	Service specifications define the capabilities, behavior, and external interfaces, but do not define the implementation.
Service Consumer	A component that consumes or uses a Service provided by another component. A component may be a provider of some Services and a consumer of others.
Service Connection	A communications connection between a Consumed Service Interface and a Provided Service Interface over a specific Binding. When a component consumes the services of a provider component, this link is known as a Service Connection (connection).
Service Data Unit (SDU)	A unit of data that is sent by a service interface, and is transmitted semantically unchanged, to a peer service interface.
Service Provider	A component that offers a Service to another by means of one of its Provided Service Interfaces
Software Module	A software module is a "quantum of executable machine code that can be programmed on a processing element". The attributes associated with software modules are: unique identifier, execution time and output rate factor. Software modules produce and consume data variables.

14. ANNEXE C – ABANDONED DATA COLLECTION MECHANISM THROUGH SELECTION CRITERIA

14.1. SELECTION CRITERIA FOR REFERENCE FRAMEWORKS

The quantitative approach allows the implementation of a standardized data collection tool in order to minimize and control possible biases in the study of each framework and to rationalize the results as much as possible in front of factual elements.

This standardized data collection tool takes the form of a table containing selection criteria. This list of criteria is actively established to be able to evaluate the framework, they are determined according software development lifecycle, compliance to standards, architectures etc.
A weight is attributed to each criterion in order to show a notion of priority, of importance of certain phases of the life cycle compared to others. We would have liked to index these weights in relation to the cost of the different phases of development of space flight software from scratch, but we did not find any relevant studies or publications dealing with these topics.

Some are inspired from the [13] or the NASA Procedure Requirements [103]. We chose to list the criteria in a structured way that fit as much as possible with [103] document structure.

The first section presents criteria on the roles, responsibilities and management of the framework. Included are institutional requirements for maintaining and advancing organizational capability in the practice of software engineering to effectively achieve the technological goals of the community, of the framework. As well as criteria on framework management activities defining and controlling the many software aspects of the framework from start to finish. This includes determining deliverables, cost estimates, tracking schedules, a roadmap, risk management, and determining the amount of support service. We can find criteria on:

- Governance;
- Open Source;
- Partnership;
- Software Classification;
- Software Life-cycle Planning;
- Training;
- Flight Heritage.

The second section presents the requirements for the software engineering life cycle. This section discusses the different phases of a lifecycle that allows for the development and maturation, the incremental evolutions of the framework and its components, from the initial concepts to the release of the framework and its possible final retirement.

Although this section does not impose a particular lifecycle model, it does support a set of criteria on the phases of the lifecycle that are close to an iterative model.
Indeed, the development of an open source framework follows an iterative, incremental logic with several million potential contributors, so some of the criteria refer to agile methods as a life cycle model.
We can find criteria on:

- Software Requirements;
- Software Design;
- Software Implementation;
- Software Testing;
- Software Release to Operation;
- Software Maintenance;
- Software Retirement.

The last, but not least, section provides criteria for software lifecycle support requirements that are not primarily targeted at a specific project phase, but generally occur with similar intensity throughout the entire lifecycle of the framework. We can find criteria on:

- Software Configuration Management;
- Peer Review and Inspection;
- Software Measurements or Indicators.

It is quite possible that some criteria are not applicable to some frameworks.

- in the case where a criterion is not applicable to a framework we will not allow ourselves to score it on the said criterion;
- in the case where a criterion is not considered at all or partially, we will give a score reflecting the estimated effort to meet it.

Criteria			Description	Coef
Software Management Requirements				
	Governance			
		Maintainers	Shall have a list of officials maintainer available to clarify who has technical authority to review contributions.	4
		Open organization	Shall be maintained by a non-profit organization with sponsors to improve the performance, reliability, enhancements and stability of the framework.	2
	Open-Source			
		Source Code accessibility	Shall have human-readable source code, broadly available at no cost.	5
		Operating license	Shall have a license as permissive as possible, in order to be as free as possible in the use of the framework (military project for example). Shall provides conditions for use, reuse, modification/improvement, and redistribution. Easily observable by an individual or organization not previously connected with its open source project.	5
		Long-term support	Shall ensure long-term support in accordance with the sponsoring and maintaining institution's strategic plan.	4
		Dependencies	All software dependencies used by the framework shall be available under a permissive Open Source license.	4
		User-community collaboration	Shall have a community where users share their experience in a centralized repository of open code on the Internet. Shall have the ability to open issue, branch and download software code, all controlled and moderated by the sponsoring institution.	3
		Domain popularity	The popularity and reputation of the framework in the space domain are assets.	2
	Training		Shall provide educational resources for the training of users/contributors of the framework. Existence of demonstration applications that provide training and guidance on how to use the framework. The provision of an SDK is considered an advantage.	4
	Flight heritage		Shall have been successfully operated in flight during previous space missions.	4
	Partnership		Shall have the ability to partner with framework managers to get support to build a demonstrator, or even to get funding.	2

	Software Classification Assessments		Shall provide information on the highest classification (A, B, C, D, E and F) applicable to the entire frame or its components	2
	Software Life-Cycle Planning		Shall enforce a policy of maintaining and executing software plans that cover the entire software life cycle. Must track actual results and performance of software activities against software plans.	1
Software Engineering Life-Cycle Requirements				
	Software Requirements			
		Formal specification	Shall use architectural rules or requirements to formally specify the software implementation.	4
		Requirements traceability	Shall have individually related requirements for implementation and evidence of verification.	1
		Well-defined semantics	Shall have clearly defined and unambiguous behavior specifications.	1
	Software Architecture			
		SPA	Shall adopt the concept of a "plug-and-play" distributed integrated network to reduce the complexity and time required for equipment avionics and integration.	5
		CCSDS SOIS	Shall be recognized as expressive by a competent authority (in this case, the CCSDS) and is being standardized as reference flight software architecture. Otherwise, the adoption of the concepts defined in the CCSDS SOIS is an advantage.	4
		Modularity (SOA)	Shall have an architecture that emphasizes separation of concerns using loosely coupled software components or independent applications. Architecture shall support reuse of software components across different missions/projects.	4
		CCSDS MOSS	Shall be compliant with the CCSDS MOSS which provides a service-oriented networked system for a high level of autonomy.	3
		SAVOIR OSRA	Shall be recognized as expressive by a competent authority (in this case, SAVOIR) and is being standardized as reference flight software architecture.	3

		TSP (Time And Space Partitioning)	Shall have the ability to perform TSP to merge multiple processing units into the main computer to ensure that deferred applications cannot affect the core software. This reduces the complexity, mass and cost of the spacecraft. It generally involves running each application in its partition, which is a virtual execution environment with no access to neighboring partitions unless they are deliberately configured.	3
	Software Design/Interface			
		Portability	Shall have architecture and application design and implementation properties that allow them to be used on systems other than the initial target system.. It generally involves software isolation and abstraction techniques (Operating Abstraction Layer, Platform Support Package).	5
		SAVOIR-SAFI	Shall implement the concepts detailed in SAVOIR-SAFI.	4
		CCSDS SEDS	Shall implement the SOIS Electronic Data Sheet (EDS) message passing mechanisms defined by in SOIS CCSDS.	4
		CCSDS SPP	Shall implement the CCSDS Space Packet Protocol, which is designed to meet space mission requirements for efficient transfer of space application data of various types and characteristics between nodes, over one or more sub-networks, and possibly involving one or more ground-to-space, space-to-space, space-to-space, or on-board communication links.	3
	Software Implementation			
		Source Code Autocoder	Shall define the approach to the automatic generation of software source code. Including validation and verification of auto-generation tools, monitoring the actual use of auto generated source code.	5
		Evolutionary complexity	Shall provide utilities to add components easily, through a logic-free and agnostic templating system. Or other utilities that help to implement a feature request (component/protocol/...).	4
		Coding rules	Shall select and adhere to software coding methods, standards, and criteria.	3
		Small footprint	Shall have minimum memory loads and CPU usage so that the framework can be used on the largest number of hardware platforms certified for space environments	2
		Linter	Shall provide primary tool for evaluating/ensuring the quality of software products. Shall use a code reviewer instantiable internally in the company to detect vulnerabilities. Shall use a code formatter instantiable internally in the company to homogenize the source code.	2
	Software Testing & Measurements			
		Unit Testing Framewok	Shall rely on a Unit Test framework, preferably external and Open Source, to ease the testing phase.	4

		Code coverage	Shall ensure that the code coverage measurements for the software are selected, implemented, tracked, recorded, and reported. Shall be measured by analysis of the results of the execution of tests.	4
		Test Suite Runner	Shall provide a tool to perform an acceptance test suite.	3
		Job Runner	Shall use a job runner with jobs provided instantiable internally in the company to automate CI/CD phases.	3
	Software Operations - Ground Support Equipment			
		Scenario Runner	Shall provide a visual interface for editing and executing test scripts/procedures.	4
		Data Visualization	Shall display streaming and historical data, imagery, timelines, procedures, and other data visualizations in one place.	4
		Data Extractor	Shall be able to record and extracts data into a file for further analysis.	4
		UI/UX	Shall be a Web-based application with responsive design to simplify support removing the need to deliver updates to multiple machines.	3
		Customizable dashboard	Shall provide a user composable layout. Extensible and flexible architecture via plugin to enable the development of new visualization, integration with telemetry sources and other new features.	3
	Software Maintenance			
		Technical documentation	Shall provide a software user guide and complete technical documentation.	4
		Workflow	Shall have a contribution process to fix issues, add enhancements, e.g. by git Pull Request.	4
		Container	Shall use a container tool instantiable internally in the company to virtualize the build chain and avoid installation issues. If not container available, it shall be able to launch the build chain/a continuous integration chain in a container	3
		Remote Development Environement	Shall have a build chain, configuration tools and GUIs that can work in a remote development environment (such as GitPod) in order to do "Read to code / Code everywhere". If not it shall be able to setup a remote development environment with the framework.	3
		Release Note	Shall provide a software version description for each software release.	2
	Software Retirement			
		Archiving	Shall identify documents and software tools to be archived, location of archives, and procedures for accessing products for removal or disposal of the framework.	2

Supporting Software Life-Cycle Requirements				
	Software Configuration Management		Shall apply configuration management throughout the software life cycle to ensure the completeness and correctness of software configuration items. Shall track and evaluate changes to software products. Shall identify the software configuration items (e.g., software records, code, data, tools, models, scripts) and their versions to be controlled for the project.	4
	Software Risk Management		Shall record, analyze, plan, track, control, and communicate all of the software risks and mitigation plans.	4
	Software Peer Reviews and Inspections		Shall have a peer review and inspection of contributions to find and eliminate defects early in the life cycle. Software peer reviews and inspections are performed according to defined procedures covering the preparation of the review, the conduct of the review itself, the recording of the results, the reporting of the results, and the certification of completion criteria.	4
	Contribution Metrics		Shall have analytical reports/indicators based on data in the framework repository to help the community gain a better overview of project activities. This helps eliminate bottlenecks in the various processes in place, improve collaboration and deliver projects faster and with better quality.	1

14.2. CLARIFICATIONS ON THE CRITERIA

14.2.1. CCSDS - Consultative Committee for Space Data Systems

14.2.1.1. SOIS - Spacecraft Onboard Interface Service

The Spacecraft Onboard Interface Service (SOIS) is defined and standardize by the Consultative Committee for Space Data Systems (CCSDS) [30].

The SOIS target system is generically declared by the CCSDS to be "all classes of civil missions, including scientific and commercial spacecraft, manned and unmanned systems".

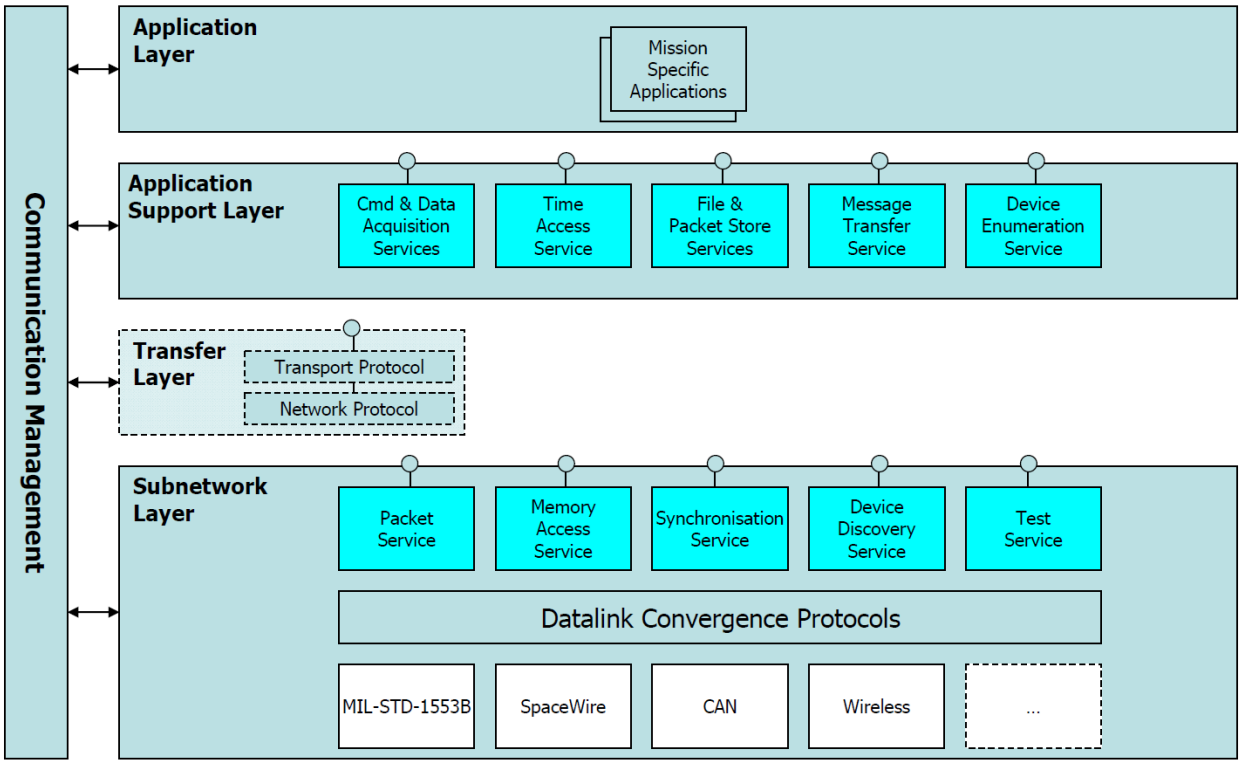


Figure 14-1: SOIS Reference Communications Architecture © CCSDS

The SOIS Reference Communication Architecture provides plug-and-play platform equipment for spacecraft by explaining how SOIS services interact together to provide a plug-and-play device, shown below:

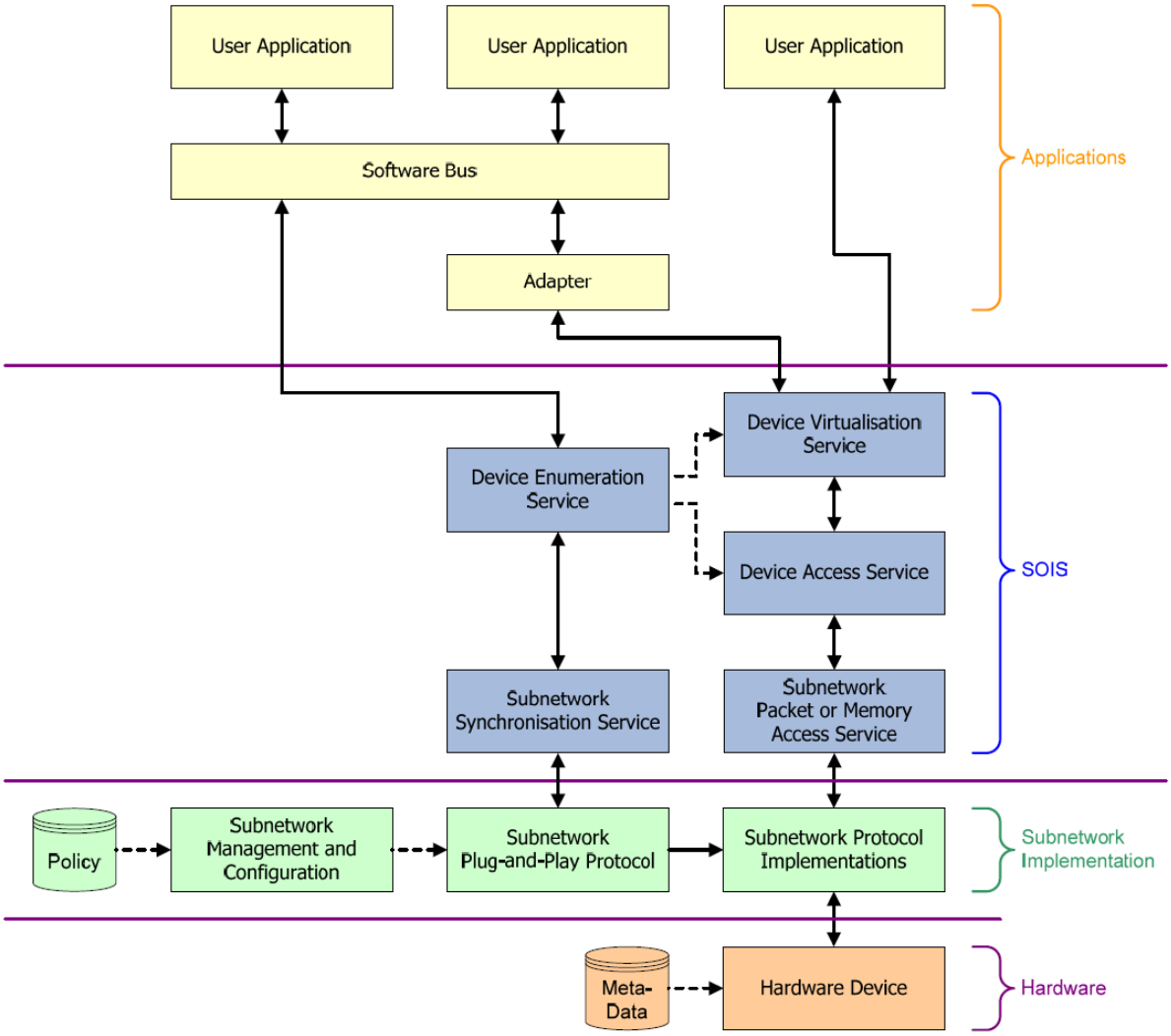


Figure 14-2: Device Plug-and-Play View of SOIS Architecture © CCSDS

14.2.1.2. SEDS – Standard Electronic Data Sheet

A SEDS, a SOIS EDS from [10] is a CCSDS standard within the SOIS framework, pursued by ESA and NASA. It is intended to be used industry-wide to describe the format of information in any data interface for any embedded device. A collection of SEDS instances may be required to describe the data interfaces of applications on an embedded computer.

A SEDS may also define a simple structural or behavioral mapping between such interfaces, such as that required to map the binary message format used by a device to the set of commands and parameters it supports.

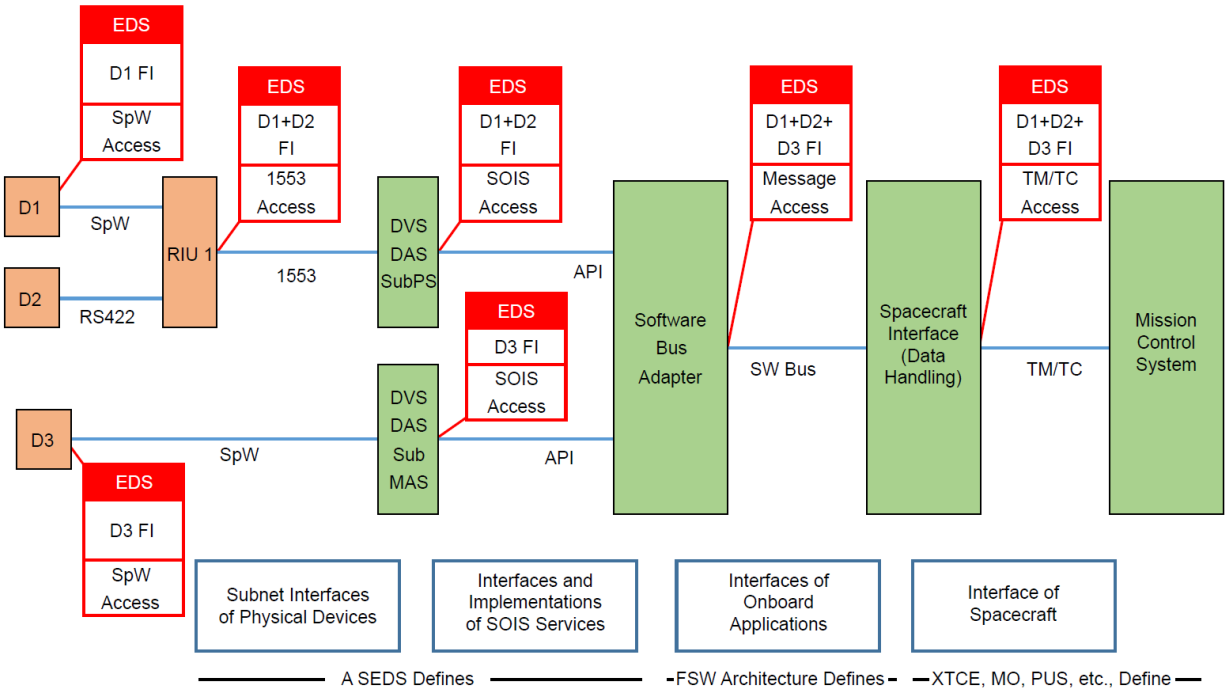


Figure 14-3: SOIS Electronic Data Sheets Describe Data Interfaces in a Spacecraft © CCSDS

Some aspects of a device data interface may correspond to standardized protocols directly usable at the application layer.

A SEDS is intended to be a machine-understandable mechanism for describing embedded components fully described in [30].

The SEDS is intended to replace the traditional interface control documents and proprietary data sheets that accompany a device and are required to determine how the device works and how to communicate with it. The SEDS could then be used for a wide variety of purposes, while ensuring consistency and completeness of information:

- generating human-readable documentation;
- specifying the interfaces with the device;
- automatically generating software implementing the relevant parts of the onboard software for the device;
- Automatically generating of device interface simulation software for use in device testing or simulation software;
- device simulation software;
- Transforming the functional interface of the device into commands and reports adapted to the processing by a command and data handling system onboard and on the ground;
- Capturing interface information for the spacecraft database.

CGI has a long history of involvement in EDS: for which it has technically led most of the EDS R&D projects and is the expert in the CCSDS and SOIS EDS working groups [104].

14.2.1.3. SPP – Space Packet Protocol

The Spatial Packet Protocol (SPP) [42] is designed as a self-delimiting medium of a data unit that contains an APID used to identify the data content, data source, and/or data user in a given enterprise. A typical use would be to transport data from a specific mission source to a mission user.[105]

Different types of data often require additional information (such as time) to fully utilize the data contained, and these parameters as well as the format of the data content must be identified, in the context of the mission, using APID.

The SPP is designed to meet the needs of space missions for the efficient transfer of space application data of various types and characteristics between nodes, over one or more subnetworks, and possibly involving one or more ground-to-space, space-to-space, space-to-space, or on-board communication links.

The SPP provides a one-way data transfer service from a single source user application to one or more destination user applications across one or more subnets.

The primary function of this protocol is the identification and encapsulation of application data to facilitate its transfer along the managed data path across the underlying subnets.

By using the space packet protocol as the application layer protocol, different network technologies, each tailored to one of the environments encountered in the system, can coexist throughout the system under the space packet protocol, and applications can be built on top of the space packet protocol independently of the underlying network technologies. On the space link, however, space packets are transferred directly over the space data link protocols without any intermediate layers, resulting in high bit rates.

SPP is used directly on top of a data link layer protocol on a space link and allows space packets to be routed through the network.

It provides routing capability across the entire network, which consists of different subnets, but some of the subnets may have routing capability tailored to routing within those subnets. In this case, the space packet protocol is used to route space packets from one subnet to another, while a local network layer protocol is used to route space packets within the individual subnets. Therefore, routing can be performed in a hierarchical manner and the space packet protocol performs routing at the application layer.

This protocol is used as the standard data unit to identify and manage application data throughout the system.

End-to-end data management is a critical issue in spatial data systems because important data may be transferred over unreliable communication links and because data may be stored in temporary warehouses and/or transferred over multiple paths.

Because the networks used for space projects consist of various types of subnetworks (on-board buses, space-to-ground RF links, the Internet, etc.), the data unit used for end-to-end management should not depend on any technology used in the subnetworks.

The space packet is frequently used as the standard data unit for:

- End-to-end management by space projects because it is neutral with respect to data transfer technologies.
- Storage and transmission, because it is neutral with respect to file systems or directory structures.

Admittedly, this protocol is slightly more complex than traditional time division multiplexing (TDM) telemetry systems. However, if data of variable length (such as compressed data) is to be transferred or if data is to be transferred at irregular intervals, it is very difficult to handle such data streams efficiently in traditional TDM telemetry and requires a complex scheme. Therefore, regardless of the size of the spacecraft, the space packet protocol is a good solution if processed data must be transferred over space links.

14.2.1.4. MOSS - Mission Operations Service Standard

The CCSDS Mission Operations Service Standard is identified as a basis for adding the building blocks of autonomy, using a Service Oriented Architecture (SOA) to facilitate the transition from the widespread "monolithic" architecture to a service-oriented networked system for new applications. This architecture allows for a high-level autonomy framework to be defined to obtain existing off-the-shelf flight software [106].

14.2.2. SAVOIR - Space AVionics Open Interface aRchitecture

SAVOIR is an initiative to federate the space avionics community and work together to improve the way the European space community builds avionics subsystems [105].

The objectives are as follows:

- reduce the schedule and risks, and therefore the cost, of acquiring avionics, and development, while preparing for the future,
- to improve the competitiveness of suppliers of avionics products,
- Identify the main avionics functions and standardize the interfaces between them so that the building blocks can be developed and reused in projects.
- Influence standardization processes by standardizing at the right level in order to
- Obtain interchangeability of equipment (the topology remains project-specific).
- define the governance model to be used for products, generic specifications,
- The definition of the interface of the elements produced within the framework of the SAVOIR initiative.

The process is intended to be applied in the context of agency invitations to tender (ITT), and throughout the procurement process and subsequent development. A particular objective is to ensure that the results of SAVOIR are exploited in future projects and products relevant to European supplier portfolios.

The main results of Insight are as follows:

- reference avionics architecture for space platform hardware and software
- a set of standard external and internal avionics interface specifications
- the definition of building blocks composing the architecture
- the functional specification of selected building blocks comprising the architecture
- the implementation of selected building blocks at the right TRL level
- Process definition and assessment.

14.2.2.1. SAVOIR SAFI – Sensor Actuator Functional Interface

This working group is responsible for standardizing the functional interface with sensors and actuators. It intends to prepare for the use of interface mechanisms based on the use of Electronic Data Sheets as defined in the CCSDS SOIS standards. This section extracts the most striking point of the document “SAVOIR-SAFI AOCS GNC Sensors and Actuators Functional Interface - Iss1 rev2a TEC-SWE/11-481/AJ” [107] applicable to our R&D.

Within requirements described in the reference document, we can note that the interfaces of GNC/AOCS equipment shall be defined within the framework of the SAVOIR-OSRA and COrDET architecture. The device manufacturers will provide an Electronic Data Sheet (EDS) that will be used to generate the Sensors/Actuators driver, which in turn will be used by the software to connect the device to the bus.

14.2.2.2. SAVOIR OSRA - Onboard Software Reference Architecture

This section is a summary of the paper “SAVOIR Onboard Software Reference Architecture SAVOIR-TN-002” [107] from SAVOIR Working group that details the SAVOIR Onboard Software Reference Architecture.

The SAVOIR Onboard Software Reference Architecture (OSRA) is comprehensive reference architecture for spacecraft on-board (Flight) software, developed as part of the SAVOIR initiative. OSRA is intended to meet the main needs for software development in a modern European context and is based on the work carried out by the SAVOIR-FAIRE working group and subsequently in a number of key ESA activities, such as COrDeT projects.

OSRA uses a component- and model-based approach to enable the development of embedded software in a more efficient and flexible way than traditional methods, without sacrificing robustness. The use of component-based software engineering (CBSE) increases software reuse, helping to shorten development times and reduce recurring costs.

It would act as a facilitator of innovation among software vendors and users because of the ability to separate critical concerns:

- Separating functional and non-functional aspects allows decisions about the dynamic design of a software system to be deferred until the components are in use, rather than at the time of their creation.
- Separating hardware and software concerns, as well as abstracting the underlying platform, allows the same components to be used in different hardware and software environments.

The OSRA is based on a three-layer architecture shown below:

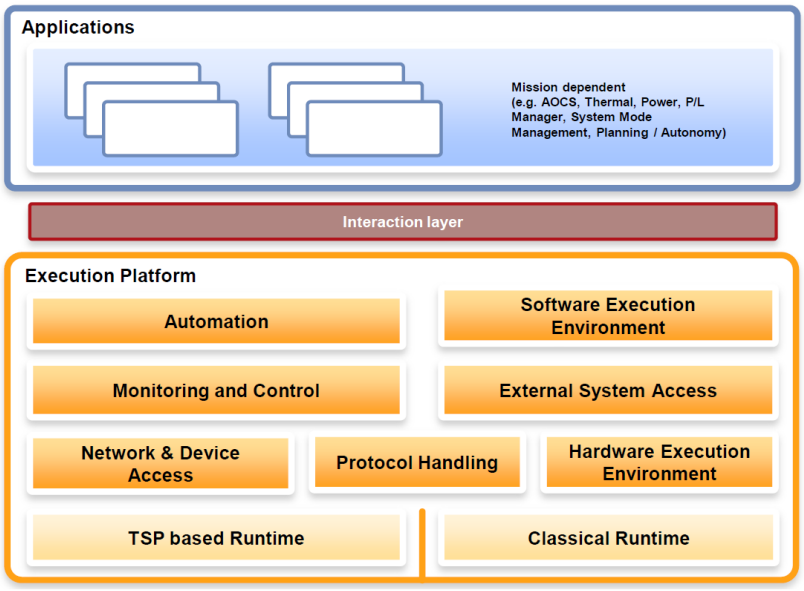


Figure 14-4: OSRA Three-Layer Architecture © SAVOIR

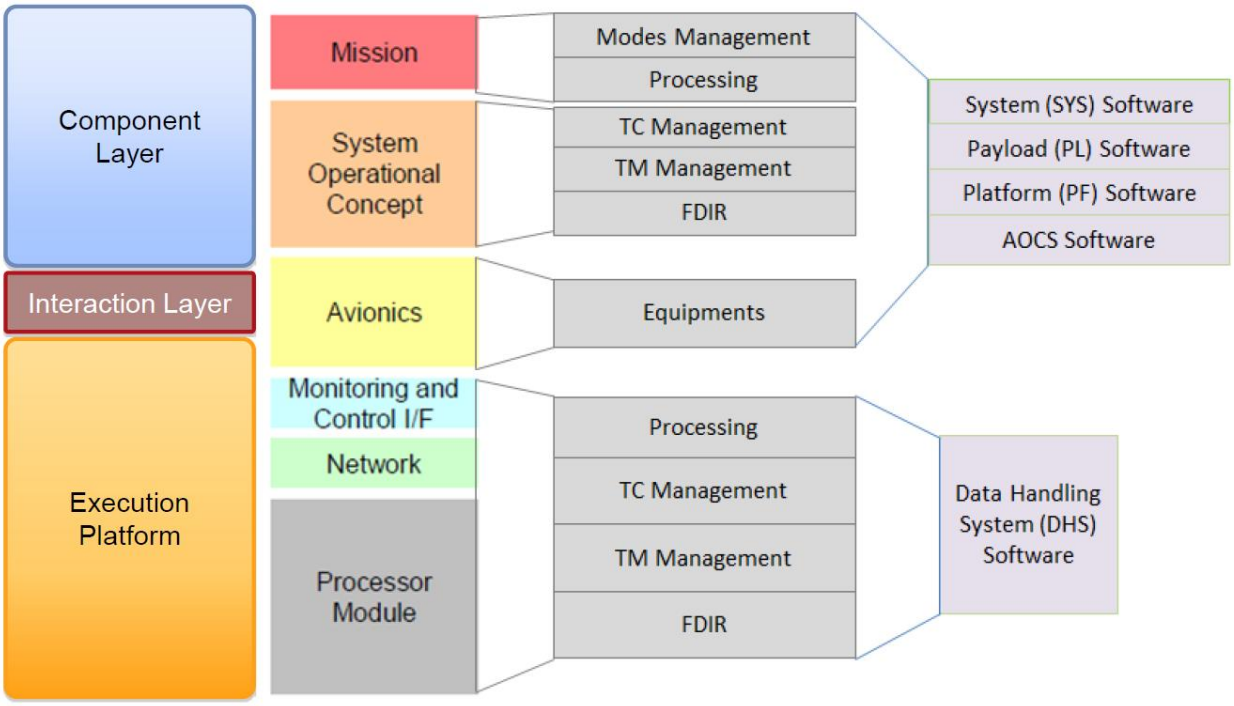


Figure 14-5: The OSRA Three-Layer Architecture and System Variability Levels © SAVOIR

14.2.3. TSP - Time and Space Partitioning

As evidenced by the SAVOIR working group document “SAVOIR Avionics Time and Space Partitioning User Needs TEC-SW/09-247/JW” [107], Time and Space Partitioning is increasingly being considered in the design of space flight software.

TSP is built using an embedded hypervisor, which is software that virtualizes a runtime computing environment [108]. This virtualization separates hardware resources into distinct runtime environments, called virtual machines or partitions. The hypervisor manages these virtual machines by allowing them to run concurrently and share resources without interfering with each other [109]. It allows system designers to run multiple operating systems and applications on a single hardware platform with varying degrees of reliability, safety and security.

Many potential benefits of embedded hypervisor are identified:

- **Increase reliability** with system monitoring, guest failure detection, and automatic shutdown and restart. By preventing guest failures, a guest can fail without propagating to the hypervisor or other guests.
- **Increases system security** by separating components into virtual environments, limiting an attacker's ability to compromise the entire system.
- **Reduce development, validation, and qualification** processes by reusing proven software. This prevents situations in which a minor change necessitates extensive non-regression testing of the entire embedded software system.
- **Use the best operating system** for the application, such as baremetal or RTOS for safety-critical software and Linux for data processing.
- **Streamline and limit the scope of security certifications** by isolating non-secure systems within security certified hypervisors.
- **Reduce hardware cost, size, weight, and power consumption** by reducing the number of hardware platforms in the overall design.

The SAVOIR working group identified necessary feature for a TSP system-based solution:

- Support shared libraries.
  A shared library can be used as long as it is re-entrant and has its local data space allocated in the partition of the calling application. It should also not be integrated with the executive but run in user mode to avoid the supervisor switching modes when a library function is activated.
- Support (and control) of shared memory.
- Access to CPU margin for sporadic and intensive calculations.
  This issue can be a criterion for defining the partitioning scenario, or can lead to a specific partitioning concept.
- Guaranteed/assured access to I/O bandwidth.
- Execution of the same application code.
- Several instances of the same function.
- Support for reloading partition software.

The SAVOIR working group outlines some future evolution of the TSP approach for spacecraft avionics:

- **Fixed versus Dynamic Scheduling**
  The use of dynamic scheduling should be considered as a way to optimize the TSP architecture if the available computing power is insufficient.
- **Dynamic Mode Dependent Scheduling**
  A reconfiguration capability to switch to a different partition schedule for different operational mission modes can be considered. It allows less critical applications to be suspended in favor of more critical functions during a particular mission phase or failure mode.
- **Multi-core processor and Processors cluster**
  The TSP solution must be able to scale to support multi-core processors and processor clusters; the transition to multi-core technology will be limited to the system executive platform vendor to configure system resource allocation.

The SAVOIR working group concludes that a TSP solution for the space domain clearly brings benefits.

14.2.4. SPA - Space Plug-And-Play Avionics

This section aims at defining the architecture of the Space Plug-And-Play avionics and is extracted from [110]. The Space Plug-and-play Avionics (SPA) used by the Air Force Research Laboratory was developed to minimize the time between specifications of a space mission and launch. SPA specifically aims to improve the robustness and integration time of other popular avionics protocols, such as MIL-STD-1553, and to create a "plug-and-play" architecture for space SM.

The SPA is inspired by the design challenges and the implementation of other plug-and-play approaches in other industries, such as:

- the Highway Addressable Remote Transceiver (HART) protocol that superimposes digital telemetry over analog current loop measurements for accurate industrial control
- LonTalk, used in industrial sensor networks
- IEEE 1451 an intelligent sensor standard that defines Transducer electronic data sheets (TEDS)
- Universal Plug-and-play (PnP), a self-organized network standard for the PC industry.

There are several motivations for such a system.

- The cost of building a spacecraft, in terms of time, money and human resources, has always been exorbitant, requiring millions of dollars and years, even decades, to reach orbit.
- The miniaturization of satellites, particularly in micro and nanosatellites, has somewhat alleviated this problem; however, even this revolution in small satellites still requires a further reduction in development and integration time.

Two notable terrestrial industries have adopted the PnP concept:

- consumer PCs with their adoption of USB and PCI (Peripheral Component Interconnect)
- Industrial factories with the adoption of PnP sensor networks through the Echelon Industrial Internet of Things, the LonTalk protocol.

SPA simply does not want to rely on these existing PnP architectures to "adapt" existing components with interfaces to communicate over USB or PCI networks; instead, a new standard is needed that is "embedded" in spacecraft components.

Although SPA is only the network management aspect of this standard, fully reconfigurable software defined radios (SDRs), programmable cabling systems, malleable signal processors and radiation-hardened components form the complete PnP image of a satellite.

The SPA standard itself aims to differentiate itself from terrestrial PnP implementations by considering specific constraints more specific to space-based MS. These include environmental constraints, synchronization, high power delivery and driverless operation.

1. Environment, processing elements in space SM shall consider radiation effects such as total ionizing dose, interlocks and single event disturbances. These can temporarily interrupt the execution of individual tasks, corrupt memory elements and even destroy processing elements.
2. Synchronization, all satellite systems must have a "unified concept of time".
3. High power, many ground-based PnP implementations offer some sort of power/data bundling; however, they are not well suited for most spacecraft power requirements (such as a 28V bus).
4. Without a driver, terrestrial PnP implementations often require drivers to work with new devices, which is not desirable for SPAs.

SPA manages network communication in a different way through its four implementations:

- SPA-U, based on USB,
- SPA-O, optically based
- SPA-S, based on SpaceWire
- SPA-I, based on I2C.

Each implementation uses this communication protocol for its physical and data link layers, but SPA extends these protocols to meet the challenge of "plug-and-play" in the space environment, including restricted hardware, higher power delivery, self-description, and fault tolerance. All of these SPA implementations are node-oriented.

Note: as described in section 14.2.1.1, if the framework implements a CCSDS SOIS compliant architecture, we can consider that it is also compliant with the Plug-And-Play concept defined in SPA.

15. ANNEXE D – CFS DISTRIBUTIONS

We will briefly present some cFS distributions that were not implemented during the study.

15.1. OPENSATKIT

The OpenSatKit "OSK" GitHub project [111] is a starter kit providing a fully functional Flight/Ground system running on a desktop computer. It was developed for several reasons:

- Serve as a distribution of the core Flight System (cFS)
- Serve as a cFS educational platform
- Provide an application prototyping environment
- Support target embedded platform evaluation and initial ports.

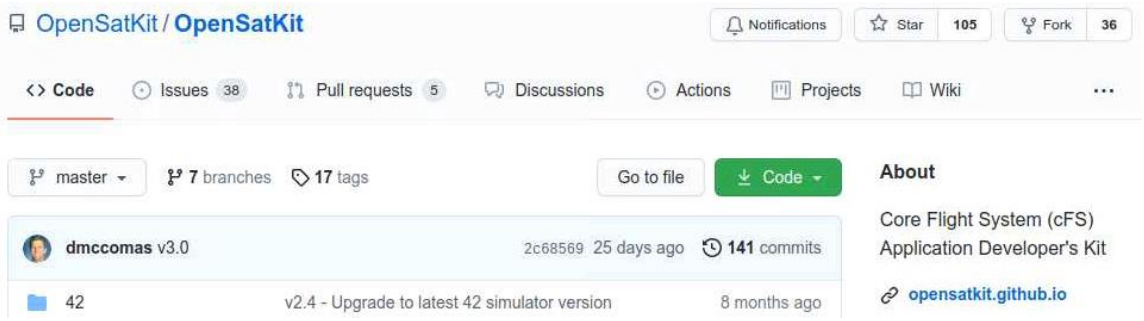


Figure 15-1: OpenSatKit GitHub Project

The OSK components are:

- Ball Aerospace's COSMOS, a user interface for command and control of embedded systems, used as the Ground System
- The cFS bundle providing FSW components
- 42 Simulator providing a simulation of spacecraft attitude and orbit dynamics and control.

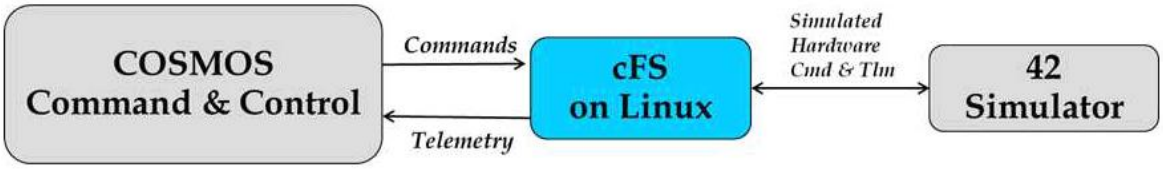


Figure 15-2: OSK block diagram © David McComas

15.2. ICAROUS

ICAROUS (Independent Configurable Architecture for Reliable Operations of Unmanned Systems) [112] is a software architecture that enables the robust integration of mission-specific software modules and highly secure core software modules for the creation of autonomous safety-oriented unmanned aircraft applications.

Figure 15-3: Icarous Logo © NASA

The set of core software modules includes formally verified algorithms that detect, monitor, and control compliance with safety criteria, avoid fixed obstacles and maintain a safe distance from other airspace users, and compute resolution and recovery maneuvers, autonomously executed by the autopilot, when safety criteria are violated or about to be violated.

ICAROUS is implemented using the cFS framework. The above-mentioned features are implemented as cFS applications that interact via a publish/subscribe messaging service provided by the cFS software bus.

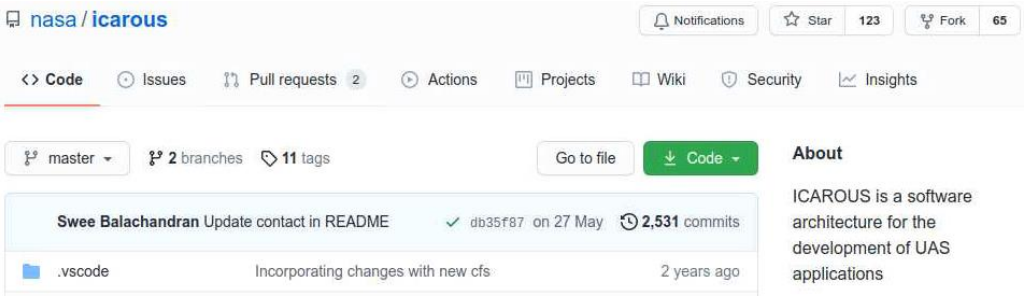


Figure 15-4: Icarous GitHub Project

15.3. NOS3 – NASA OPERATIONAL SIMULATOR FOR SMALL SATELLITES

NASA's Small Satellite Operational Simulation (NOS3) [113] was developed by the Jon McBride Software Testing and Research (JSTAR) team in response to the STF-1 mission.

NOS3 allows multiple developers to create and test flight software with simulated hardware models. The flight software is unaware that it is not being run in space, as it receives the same data as during nominal operations.

Figure 15-5: NOS3 logo © NASA

NOS3 leverages NOS middleware to include a full suite of CubeSat simulations on a virtual machine (VM) platform. It integrates a variety of components such as:

- NASA's cFS
- Hardware simulators
- Vagrant
- COSMOS
- OIPP (Orbit Inview & Power Prediction)
- "42" simulator.

Each software component is described in more detail below:

- NOS, the core technology of NOS3, is a solution developed by JSTAR to simulate hardware buses as software-only buses. This component provides connectivity between the flight software and the simulated hardware components.
- NASA's cFS is a reusable, project-independent, open source flight software framework that serves as the base system for STF-1 flight software development.
- Hardware simulators are fully customizable software models of a specific piece of flight hardware that often focus on the inputs/outputs of the device.
- Each of these models serves as virtual hardware to provide the flight software with an accurate representation of its data.
- Vagrant allows a computer to configure the virtual machine needed to run the applications associated with the NOS3 suite.
- COSMOS is open source ground system software developed by Ball Aerospace that is used to provide command and control of the flight software.
- OIPP is a planning tool developed by NASA IV&V team members that allows the ground station to know when the satellite will be in view, as well as when the satellite will be in sunlight.
- This allows the IV&V team to plan the power usage and communication times of STF-1.
- "42" is a spacecraft attitude and orbital dynamics visualization and simulation tool developed by NASA.
- "42" provides magnetic field data and position data as inputs to magnetometer and GPS simulators.

NOS3 has been fully developed and is open source, which allow JSTAR to make this software available to everyone.

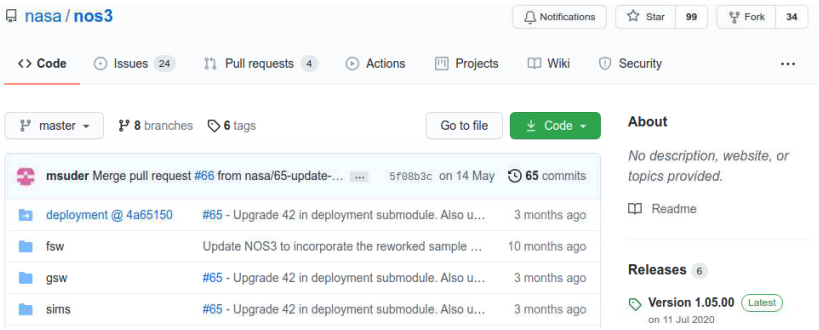


Figure 15-6: NOS3 GitHub project

16. ANNEXE E – THIRD-PARTIES FRAMEWORK / GROUND SOFTWARE

16.1. OPENMCT

OpenMCT (Open Mission Control Technologies) [114] is a next-generation mission control framework for data visualization on desktop computers and mobile devices. It is developed at NASA's Ames Research Center at Silicon Valley in collaboration with NASA JPL.

Figure 16-1: Open MCT maintainers © NASA

It is used by NASA for spacecraft mission data analysis and experimental rover system planning and operations. As a generalizable, open source framework, Open MCT could serve as the basis for creating applications for planning, operating, and analyzing any system that produces telemetry data.

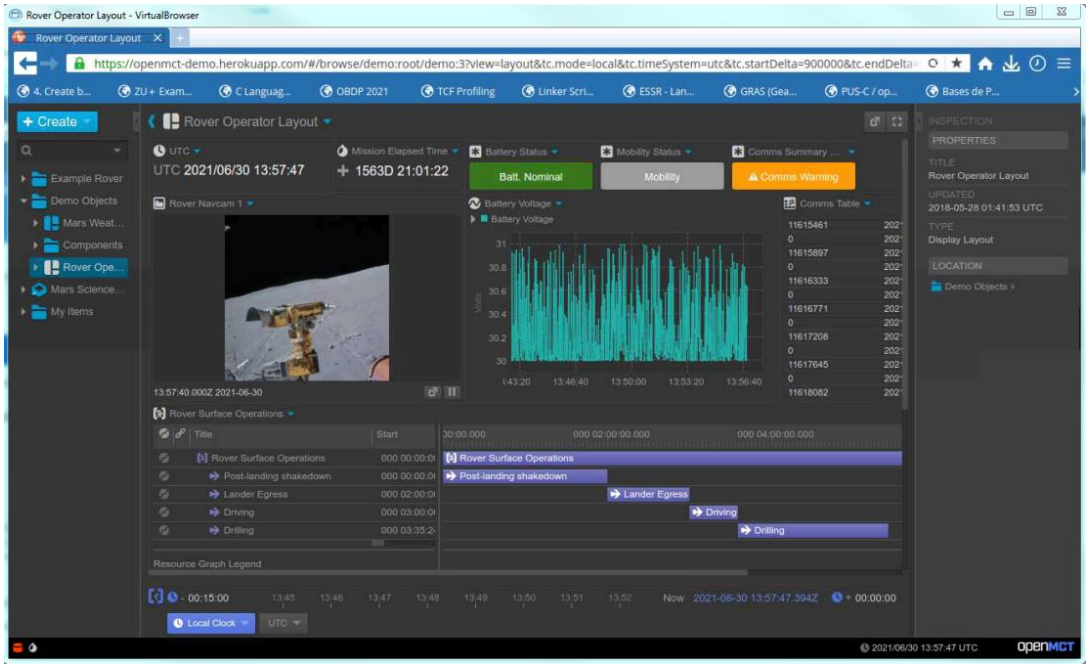


Figure 16-2: Screenshot of the Open MCT live demo

Open MCT is designed to meet the rapidly changing needs of mission control systems. At NASA, the requirements for Open MCT are driven by the need to support distributed operations, anywhere data access, data visualization for spacecraft analysis that spans multiple data sources, and flexible reconfiguration to support multiple missions and operator use cases. Open MCT consolidates many of the functions of mission operations, eliminating the need for operators to switch between applications to view all the necessary data.

Open MCT does not have a native command interface but the YAMCS framework, which supports command, works well with Open MCT. A YAMCS plugin for Open MCF has been developed to connect Open MCT to a YAMCS backend.

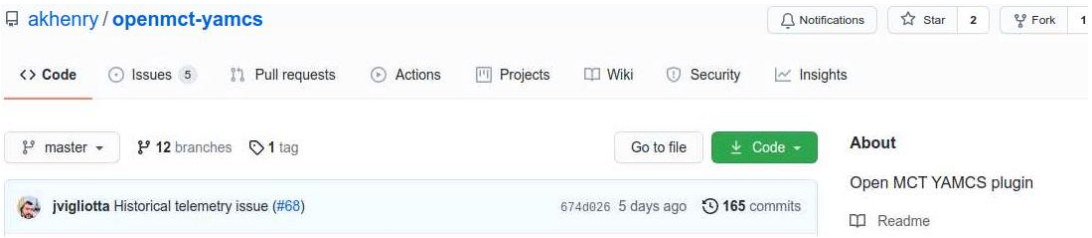


Figure 16-3: YAMCS plugin for Open MCT

16.2. YAMCS

Yamcs [115] is an open-source software framework developed by Space Applications Services, an independent Belgian company, with a subsidiary in Houston, USA. It provides command and control of spacecraft, satellites, payloads, ground stations and ground equipment.

Yamcs comes with built-in support for industry standards like CCSDS, CFDP and XTCE and includes a default set of configurable TM/TC connectors over TCP and UDP.

Yamcs Server, or short Yamcs, is a central component for monitoring and controlling remote devices. Yamcs stores and processes packets, and provides an interface for end-user applications to subscribe to real-time or archived data. Typical use cases for such applications include telemetry displays and commanding tools.

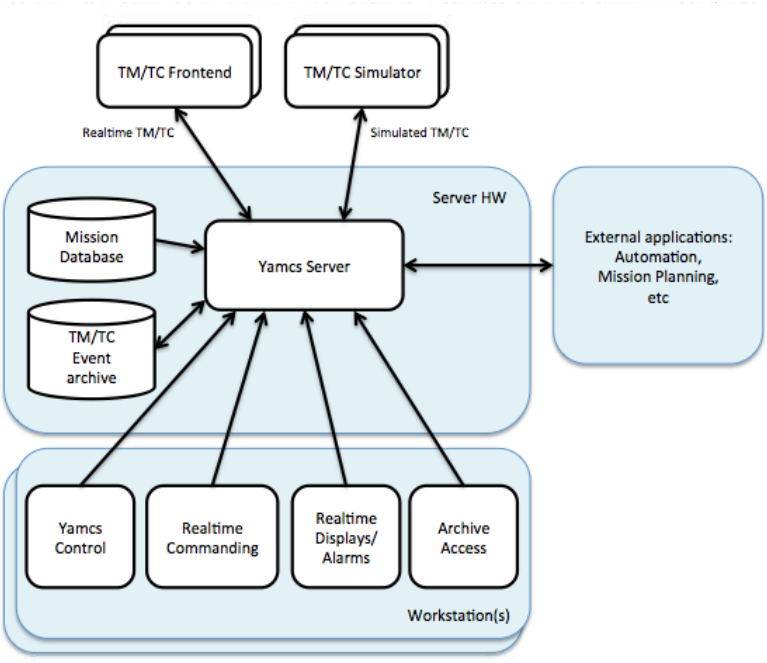


Figure 16-4: Yamcs server architecture © Yamcs

The front-end application to interact with the Yamcs server is usually implemented with Open MCT or Grafana.

As we have already put a screenshot of Open MCT above, we will only put a screenshot of Grafana below.

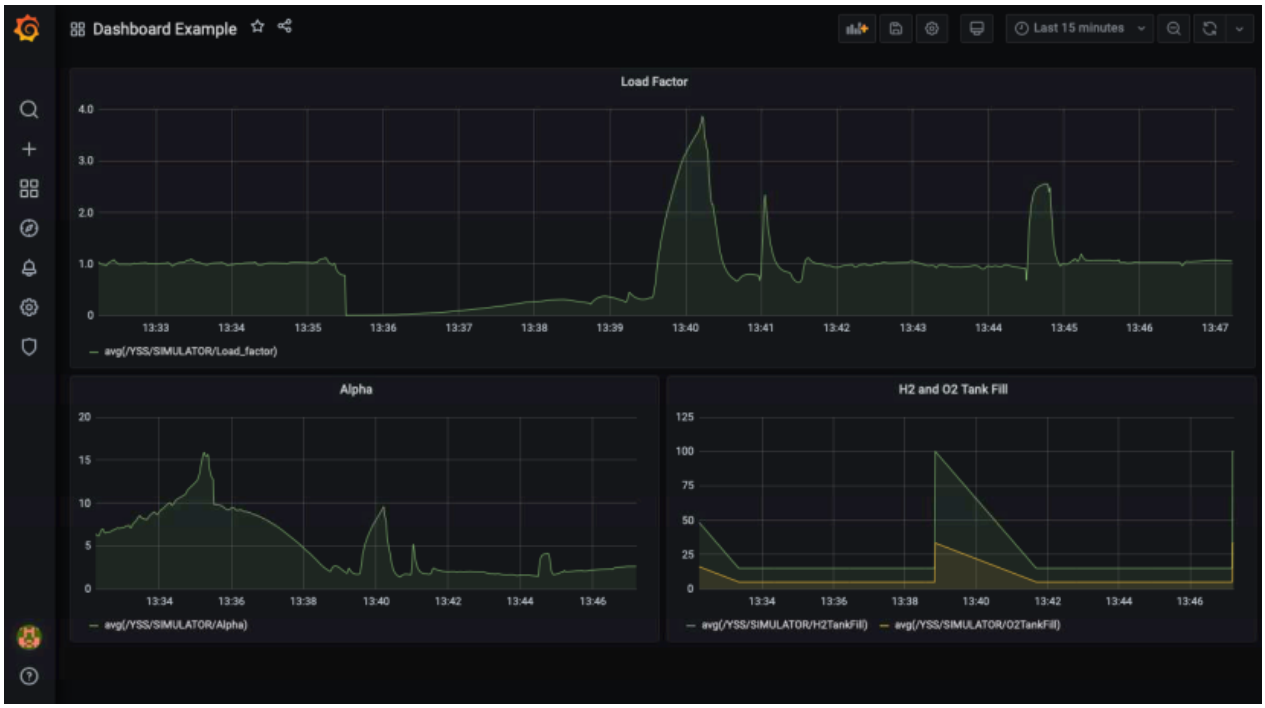


Figure 16-5: Grafana interfaced with Yamcs server

17. ANNEXE F – SEDS COMPLIANCE OF CFS

The full integration of SEDS within cFS is an ongoing, with an upcoming release expected. It does not appear to have been performed previously, particularly because whether or not SEDS should be used between applications is decided at the mission level.

Meanwhile, there is the "cfe eds framework" [98] distribution of cFS that shows how SEDS can be used, which can be useful for learning concretely what SEDS is.

This SEDS implementation does not use the "ccsds.sois.seds.xml" file available at the SANA website [116] to define a non-normative collection of definitions that can reduce the number of definitions in an EDS.

The basic type definitions of cFS are standardized in the "**BASE_TYPES"** *Package* with the *PackageFile* "base_types.xml". In this file we can see that:

- it takes all the type definitions of stdint.h which defines the standard ISO C99:7.18 integer type
- The boolean type takes only one (1) bit
- The standard for encoding and precision of floats is IEEE754_2008_single
- The encoding and precision of doubles is IEEE754_2008_double.

These definitions apply only to all internal data, i.e. not coming out of the software.

The command and telemetry formats are defined according to the CCSDS Space Packet protocol standards [42]. These definitions are defined in the "**CCSDS**" *Package* with the *PackageFile* "ccsds_spacepacket.xml".
These two *PackageFiles*, "base_types.xml" and "ccsds_spacepacket.xml" allow defining common/generic types for all components of the cFS framework.

In addition to these *PackageFiles*, the interfaces of the five (5) cFE services (ES, FS, SB, TBL, and TIME) and of the applications are defined in SEDS format.

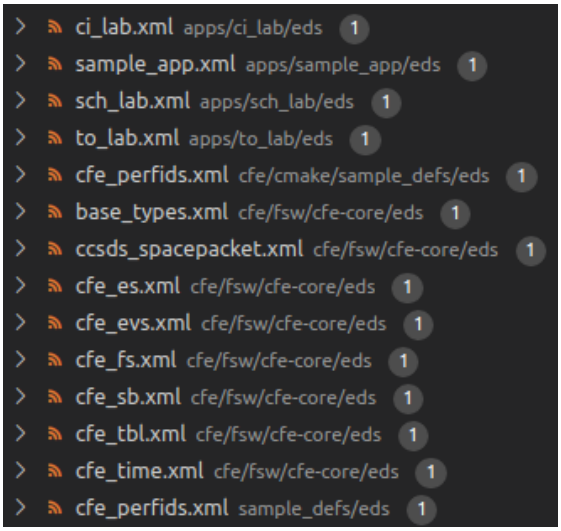


Figure 17-1: SEDS files in cfe-eds-framework

The framework uses the open source **EdsLib** library available on NASA's GitHub account [99].

18. ANNEXE G – SECT – SAVOIR EDS COMMON TOOLING

The SAVOIR EDS Common Tooling (SECT), a Java library and command line application, is owned by ESA. It was developed by SCISYS (ex-CGI) as part of a series of research projects, the most recent of which was SAVOIR EDS. It has been used as a reference implementation in the CCSDS SOIS EDS standard framework for interoperability tests. It is available on the ESSR - European Space Software Repository [117] under an ESA Community License [118].

In contrast to NASA's EdsLib tool, which is focused on generating code and other artifacts for cFS, SECT is more focused on EDS itself, doing architecture-independent things like validation, transformation, interpretation, and document generation.

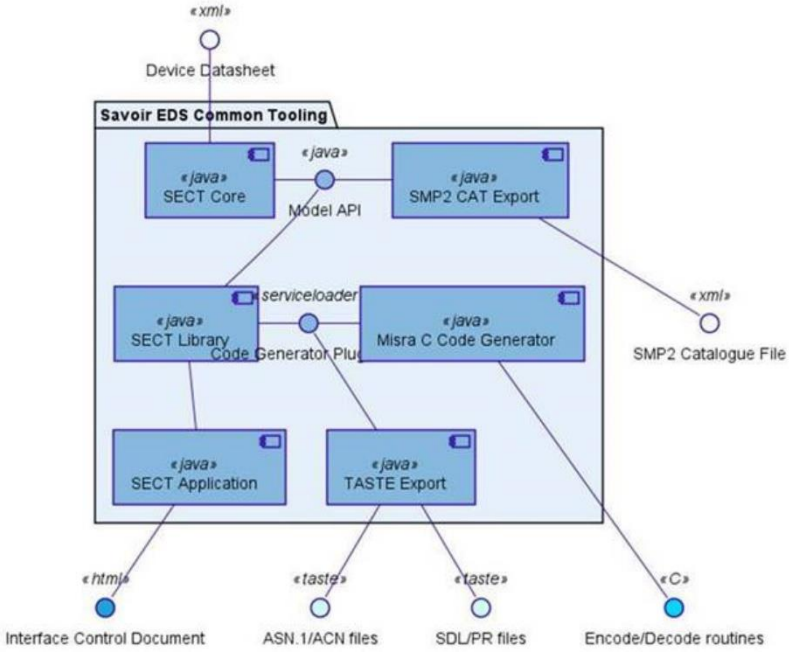


Figure 18-1: SAVOIR EDS Common Tooling © SAVOIR

Both tools could be easily integrated and used at the outset of a project; however, the choice of one or the other as the foundation for custom development would be heavily influenced by the nature of that development, including the language used. However, for non-trivial SECT use, it is strongly recommended that a working relationship be established with either ESTEC or SCISYS. Because it is a private company, the latter would imply some sort of consulting or support contract.

For the behavioral aspects of EDS communication models, the SECT tool supports both code generation (for use in resource-constrained environments such as FSW) and an EDS interpreter that directly executes the CCSDS SOIS EDS specification without requiring a compilation step. This tool can generate ASN.1 message definitions and the accompanying ACN encoding specification.

The SECT tool supports both code generation (for use in resource-constrained environments such as FSW) and an EDS interpreter that directly executes the CCSDS SOIS EDS specification for the behavioral aspects of EDS communication models without requiring a compilation step. This tool can generate ASN.1 message definitions as well as the associated ACN encoding specification.

The project is currently private because ESA prefers to maintain control over who has access to it. So we will have to ask them for access, and it appears you'll have to contact Marek Prochazka. There are no plans to make the tool available on the ESA's GitHub. The source code is available On the ESA Gitlab [119].

Note: to set up the development/test environment, the tool employs Vagrant [120] rather than Docker. The user manual, which can be found in the project's docs section on Gitlab, contains instructions.

END OF DOCUMENT