IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version July 04, 2019.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

An architecture-tracking approach to evaluate a modular and extensible flight software for CubeSat nanosatellites

CARLOS E. GONZALEZ[1], CAMILO J. ROJAS[1], ALEXANDRE BERGEL[2], and MARCOS A. DIAZ[1]

[1]Space and Planetary Exploration Laboratory (SPEL), Electrical Engineering Department, Faculty of Physical and Mathematical Sciences, University of Chile (e-mail: spel@ing.uchile.cl)

[2]ISCLab, Department of Computer Science (DCC), Faculty of Physical and Mathematical Sciences, University of Chile (e-mail: abergel@dcc.uchile.cl)

Corresponding author: Marcos A. Diaz (e-mail: mdiazq@ing.uchile.cl).

This work has been partially supported by the grants Fondecyt 1151476, Anillo ACT1405, Fondequip EQM150138 and CONICYT-PCHA/Doctorado Nacional/2016-21161016.

ABSTRACT Delivering better flight software is an important concern to improve CubeSat missions success. It has been identified as a key element to enhance team collaboration, increase reusability, reduce the mission risk and facilitate development and operation of new mission concepts, such as satellite mega constellations. An appropriated fight software architecture represents the functional and non-functional requirements, and guides the development. Therefore, to achieve the expected software quality the architecture should be closely monitored during the entire software life cycle. However, ensuring that a flight software for a spacecraft embedded system closely follows the proposed architecture and addresses the set of non-functional requirements is a difficult and non trivial problem. Motivated by requirements commonly described in previous CubeSat missions, in this work we present the design and implementation of a *flight software architecture* based on the *command design pattern*. We also present an *architecture tracking methodology* to verify and control the flight software quality criteria during the development process through the use of *graphical software analysis tools* and agile programming techniques. This automatic software analysis tool was developed using Git, Jenkins, Moose, and Roassal, and has been applied in the SUCHAI series of nanosatellites to evaluate the impact of the architecture verification during the development history. The implemented flight software and the verification tools has been released as *open source* platforms and are available for the CubeSat community.

INDEX TERMS cubesat, embedded software, flight software, nanosatellites, software architecture, software quality, software visualization, open source

I. INTRODUCTION

Historically, space missions and satellites have been developed either by space agencies or large companies. However, this context is changing thanks to the emergence of an standardized type of nanosatellite, the CubeSat [1]. CubeSats were conceived as an educational tool where students could develop, through hands-on experience, a complete mission, including design, construction, launch, and operation of a satellite in the time frame of a college degree. The results obtained since its conceptions attracted the attention of entrepreneurs, which founded small companies not only to offer CubeSat parts or platforms, but also for services. In contrast to classical satellites, space missions based on CubeSats are developed in extremely flexible work environments [2]. In this new and particular scenario, developers may be inexperienced and can easily enter and leave the project. Also, the agile philosophy of CubeSats commonly requires a simple procedure for including and removing payloads, ideally implying minimal changes to the system. The flight software is one of the critical elements to ensure the mission success in this context because the flight software implements most of the functional requirements and its complexity is related to the mission risk [3], [4]. Furthermore, the flight software also enables repeatability and scalability of nanosatellite missions to constellations consisting of hundreds or thousands of units.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

Table 1. Review of flight software architectures used in CubeSat projects

Project	Architecture details	OS supported	Language	Hardware supported	Source code	License
PilsenCUBE [5]	State machine	N/A	C	NXP LPC2148	No	N/A
Delfi-n3Xt [6]	Layered. State machine. State machine in the application layer.	N/A	C	TI MSP430F1611	No	NI
RACE, ARMADILLO [7]	Layered. Component based modules. State machine to execute modules functionalities in the application layer.	GNU/Linux	C, C++	NXP LPC3250	No	N/A
UWE-2 [8]	Centralized. Modules controlled by a central module.	uClinux	C	Hitachi H8	No	N/A
Kysat [9]	Layered. Component based modules. Centralized tasks organization.	Salvo RTOS	C	TI MSP430F1611	No	N/A
Kysat-2 [10]	Layered. SPA distributed messaging system.	SPA middleware	NI	SL 8051F930	No	N/A
PolySat [11]	Modules separated in processes, inter process communications with UDP sockets	GNU/Linux	C	Atmel AT91SAM9G20	No	N/A
ESTCUBE-1 [12]	Layered. Modules as independent tasks.	FreeRTOS	C	STM32F1	No	N/A
WinCube [13]	Layered. Modules as independent tasks.	Salvo RTOS	C	TI MSP430F169	No	N/A
Asundi *et al.* [14]	Distributed. Functionalities distributed across two microprocessors.	NI	NI	MSP430, TI C6000	N/A	N/A
3Cat-1 [15]	Layered. Two high level layers: System Core and Process Manager. Modules are threads with messaging system and task scheduler.	Linux	Prolog	AT91SAM9G20	No	N/A
NUTS [16]	Layered. Service oriented. Inter task and inter processor communications with CSP.	FreeRTOS	C	Atmel AVR32UC3, SAMV71	Yes	NI
CubETH [17]	Component-based model, verified and validated with BIP framework.	N/A	C, C++	SL EFM32GG880	No	N/A
EQUULEUS, PROCYON [18]	Layered. Command Centric Architecture (C2A).	N/I	C, C++	N/I	No	N/A
NASA Core Flight System [19]	Layered. Service oriented. Publisher-Subscriber inter task messaging system.	GNU/Linux, VxWork, RTEMS	C	x86, RAD750, MCP750, Coldfire, and others	Yes	NASAs Open Source Agreement
Kubos [20]	Layered, N-thier architecture. Service oriented.	GNU/Linux	Rust, Python, C	BeagleBone Black (Cortex A8), ISIS OBC (ARM9)	Yes	Apache 2.0
Brightascension GenerationOne	Layered. Component-based framework. Component generator using XML definitions.	GNU/Linux, FreeRTOS, RTEMS	XML, C	Nanomind A712(ARM7TDMI), Clayspace OBC (FPGA based ARM Cortex M3), BeagleBone Black (Cortex A8), TI MSP430, Vorago VA10820, Xiphos Q7	No	Commercial

N/A: Not applicable. NI: No information available

Significant efforts have been developed to provide better flight software for nanosatellites. Table 1 summarizes publications describing the flight software of CubeSat missions. Most of the analyzed solutions separated the software in layers to encapsulate different abstraction levels. This decision is accompanied by the use of an operating system (OS), where third-party solutions such as GNU/Linux or FreeRTOS are preferred. On the other hand, CubeSats are using a variety of on-board computers (OBC) from 16-bits microcontrollers to modern ARM-Cortex microprocessors capable of running GNU/Linux.

Undoubtedly, the election of the processor, the OS, and the programming language are closely related, and this decision directly affects other mission variables. On one hand, if the flight software requires a high-level programming language or entirely depends on features available only in GNU/Linux, then a more powerful processor is needed, which impacts the power consumption. On the other hand, using an OS for embedded systems, can save power at the cost not only of the processing capabilities, but also limiting the availability of developers, since they are required either to know or learn a low-level programming language.

Based on the experience of the Cubesat missions listed in Table 1, modularity, extensibility, flexibility, robustness and fault-tolerance have been identified as the main features of the flight software for nanosatellites. Most of the time the upper abstract layer is the result of architectural decisions to provide these quality characteristics, often considered as non-functional. Thus, the reviewed solutions can be grouped in: state machines [5]–[7], centralized architectures [8], [9], [11], [12], distributed architectures using messaging systems [10], [13]–[16], and formally verified architectures [17], [21]. Modularity is a common concern for flight software developers, but the definition of a module

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

varies from one solution to another. In Schmidt *et al.* [8] and Hishmeh *et al.* [9] modules are defined as a set of functionalities while in Manyak *et al.* [11] are defined as tasks or threads. Flexibility or extensibility are also common goals in this context. State machines type solutions offer a simple and clean implementation of the flight software once the satellite's functional requirements are well defined; however, there are no metrics about how a change in the requirements, during the development process, affect the evolution of the states and transitions. Component-based and service-oriented architectures are more flexible solutions to support an incremental development or changes in requirements, depending on how the components/services are orchestrated. Messaging systems add more flexibility and less coupling between modules compared with centralized architectures, especially if a publisher-subscriber pattern is implemented [19] in which subscriber modules can be added or removed without affecting the entire system. Distributed systems are a common solution to provide robustness and fault-tolerant capabilities by physically duplicating the computational resources.

Nowadays, it is possible to find ready to use flight software solutions for CubeSats being the NASA Core Flight System (cFS) [19], the Kubos initiative [20], and the Brightascension GenerationOne some of the alternatives. Except for NASA cFS and Kubos, all the reviewed works are not open source or do not report details of the actual implementation. Without this information it is challenging to evaluate software quality criteria beyond the design phase; moreover, as far as we are aware of, there is not any standard and low-cost methodology to verify software quality criteria for space systems neither in an agile fashion nor in real time. The guidelines provided in Araguz *et al.* [22] represent an important step in this direction. The cFS developers have used, among others, unit testing and graphical tools to verify the architecture and quality of the software [23], [24] but these tools are not continuously integrated into the development process in a way that might allow real time monitoring of the architecture after contribution of different developers.

The numbers of satellites proposed for the coming constellations are unprecedented. It is critical that current and future flight software solutions facilitate mass production of satellites and operation of a large number of spacecrafts. Therefore, it is imperative that the declared features of the flight software, such as modularity, flexibility and extensibility, could be evaluated in agile manner, during the development and integration of the software in each mission.

1) Our contributions

In this article, we present an *architecture-tracking approach* to achieve, evaluate, and maintain the quality of the flight software during its development. The proposed methodology assumes that if we define a clear software architecture that satisfies our set of functional and non-functional requirements, then by validating that this architecture is being followed at any point of the development process, the deployed software will also satisfy the expected quality attributes. Moreover, if we can track the architecture of the implemented software during its development history, this supervision may prevent quality deterioration, thus reducing the probability of software errors and reducing mission risk.

To accomplish this, we propose a *flight software architecture based on the command design pattern*, which is described and implemented in detail. This architecture was designed with the set of non-functional and functional requirements in mind, resulting in a highly modular, extensible, and reusable solution. This solution is evaluated using an architecture-tracking approach. Therefore, we developed a set of *visualization tools to extract the architecture automatically* from the source code. The visual analysis tools were integrated into the software development process to evaluate if the proposed software architecture is being represented by the implemented source code.

The methodology was evaluated by studying the quality of the flight software developed for the SUCHAI-1 [25], launched into space in 2017. In addition, the methodology has been used in the development of the other nanosatellites of the SUCHAI program, which includes the SUCHAI-2 and -3 nanosatellites [26]. The SUCHAI flight software had to satisfy the identified requirements including producing and operating a large number of vehicles.

These contributions represent an interdisciplinary effort that combines experiences in fields such as software engineering, embedded systems development, and space systems design. In summary, our article makes the following contributions:

- *Flight software quality monitoring.* It presents a quality monitoring methodology for spacecraft flight software. Our approach is based on principles of agile programming and *software visualization* techniques but adapted to meet the constraints of critical/autonomous embedded systems. The tools are offered as an open software platform. To show the properties of the methodology, we use the SUCHAI flight software development as a case of study, presenting the design and implementation in detail.
- *Requirement analysis.* It revises and summarizes the state-of-the-art in terms of software requirements for on-board nanosatellite flight software systems. This analysis is used not only to present the motivations of the SUCHAI flight software but most importantly to design and implement the evaluation tools according to mission requirements since in the end, these are the main goals to be satisfied by the flight software.
- *Flight software architecture*. It presents a flight software architecture for nanosatellites, based on the *command design pattern*, that satisfies these quality requirements. This software is also an open source project with flight heritage (SUCHAI-1), which is currently being adapted to the coming missions of the SUCHAI program. We use the implemented flight software to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

exemplify the use of the *architecture-tracking* method and how changes are identified in the tracking tool.

2) Outline

The article is organized as follows: Section II analyzes the requirements that commonly have to be satisfied when developing the flight software for nanosatellites. Section III proposes an architecture for flight software that meets the identified requirements and its implementation key points. Section IV describes the techniques used to validate the architectural rules over the implemented flight software, including resultant visualizations. Section V shows how to extend the software with new functionalities and analyzes the effects of these changes in the global architecture. Additionally, in Section VI we discuss about the usage of this software architecture in the SUCHAI series of nanosatellites, including pros and cons, and the usability of the visualization tool. Finally, Section VII summarizes the main conclusions of the work and presents possible avenues for further extensions.

II. REQUIREMENTS

We based our flight software decision on previous works that collect and analyze in depth the flight software features and requirements such as listed in Table 1. In this section, we explicitly present a summary of the desired flight software characteristics, both non-functional and functional requirements, in the context of agile CubeSat missions, high flexibility of mission goals, and a constellation of large numbers of nanosatellite. We also present the reasons that motivated the requirements and architectural decisions behind the implementation of the SUCHAI flight software.

A. NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements refer to the quality attributes of a system [27]. Beyond the functional aspects of a satellite mission, it is necessary to define some design guidelines that will affect the architectural decisions, especially in the context of agile, flexible and fast-growing CubeSat projects. This context makes non-functional requirements more relevant not only in the design phase, but also in the development process. Based on the literature review of a series of CubeSat missions and flight software, we detail the quality attributes considered in this work and the motivations behind this selection.

Extensibility: An extensible software is required to support iterative development of CubeSat projects. The iterative development is crucial in low-cost and fast-delivery projects to deal with uncertainty in mission deadlines, launch opportunities, component shipments or budget limitations. In the lean development strategy for CubeSat projects, it is highly desirable to have a working platform supporting the Minimum Viable Product (MVP) from the beginning of the project [2], [28]. A primary CubeSat platform consists of a bus that includes a command and data handling system (C&DH), energy power system (EPS), the communication system or transceiver (TRX) and the attitude determination and control system (ADCS). If the base bus can orientate itself, acquire data from a set of sensors, send this data using the transceiver, and send a periodic beacon, then the platform may be considered an MVP. To obtain the final product a series of incremental modifications to the flight software is required. These modifications should not affect the functionalities of the base system and may be reverted at any time with minimum effort. This is the definition of an expandable or extensible software [29]. For example, in the Kysat mission, four flight-ready revisions of the flight software were released before the final launch [9].

Modularity: A modular flight software can facilitate team collaboration and adds flexibility to integrate or remove payloads. Team collaboration in CubeSat projects implies the coordination of technical staff having a varied background. As described in Bouwmeester *et al.* [30], most CubeSat missions were originally related to technology demonstration or educational projects, which means that not only software developers, but also scientists, engineers, and even students can collaborate with the programming [9], [12], [16], [30]–[33]. Development groups focus on a specific component, subsystem or payload of the mission. Furthermore, with a standardized nanosatellite bus plus a modular flight software architecture, we can obtain a flexible platform to integrate a variety of payloads. Also, modularity facilitates the understanding of the code and debugging process, since functionalities are well defined in specific modules.

Reusability: Hardware tends to evolve more frequently than the software architecture and overall logic, especially across different missions [3]. Processors and peripherals evolve rapidly; moreover, newer and better technology is constantly available for the same price of previous hardware. In contrast to classical satellites, CubeSats projects are more prone to adopt newer technologies with limited in-space heritage. For example, the SUCHAI-1 mission, the first University of Chile CubeSat project that started in 2011, used a Microchip™ PIC24 16-bit microcontroller while the SUCHAI-2 CubeSat (started in 2015) is using an Atmel™ AVR32 32-bit microcontroller. These two embedded system platforms largely differ in terms of computation resources, hardware architectures, and development tools. They also differ compared with an x86 or a modern ARM machine capable of running GNU/Linux. Some CubeSat missions are using OBC platforms which support GNU/Linux Manyak *et al.* [34], and Javanainen *et al.* [31] proposes that using GNU/Linux instead of FreeRTOS in the Aalto-1 flight software facilitates the inclusion of non-embedded system experts in the project. However, computational resources are limited in other missions; for example, the NUTS mission preferred a FreeRTOS-based flight software [16]. Therefore, the flight software should work on both platforms. Many

University-CubeSat projects have evolved into CubeSat programs. Some of the universities that conduct CubeSat development on regular basis are Cal Poly [34], University of Tokyo [35], [32], Aalborg University (AAUSAT) [36], University of Kentucky/Morehead University (Kysat) [10], TU Delft (Delfi, DelFFi) [32], and University of Chile (SUCHAI) [26] to name a few. These programs developed flight software solutions that have been reused and enhanced along missions with different levels of difficulty. This will also be the case of future large nanosatellite constellations that will be developed and maintained for years. While the constellation objective may not change over time, these projects may face several technological changes. Thus, it is desired that a flight software should be portable and reusable by design [29].

Reliability: CubeSat nanosatellites, like any other space mission, are critical systems that must operate autonomously for years and in most cases, without a way to fix any hardware or software failure. For this reason, the flight software should be extensively tested before launching [7], [9], [37], and component redundancy is implemented to deal with hardware failures [10], [14], [16], [36], [38]. However, whole coverage testing is most of the time impracticable, and there are other methods to improve mission reliability. For example, it is crucial to reduce failure points as much as possible. Some coding recommendations to avoid errors include limiting the usage of mutexes, dynamic memory allocations and operations with pointers [39], [40]. A clear data or message path, and clear application logic can also help to identify and trace errors in the software. The integration of quality verification tools into the flight software development process could improve reliability by guaranteeing that the architectural decisions are followed without waiting for final testing phases to identify these kind of failures [23], [24].

Scalability to constellations: CubeSat standard has proven to be a cost-effective manner to reach space. Besides, the standardized satellite has shown signs of being a disruptive technology concept, exponentially acquiring new capabilities. These capabilities have made available a whole new spectrum of missions and space service models based on constellations of hundreds or even thousands of satellites. Agencies and companies are proposing, developing and even already operating large constellations of nano or small satellites [2], [41]–[43]. In contrast to classical satellites, standardized satellites can be delivered to space in short times. However, thus far they can be produced by few or at the most by tens per month. To make a reality the envisioned constellations they need to be produced by hundreds per month with higher sophistication and reliability. On the one hand, the flight software must consider the mass production of satellites and the operation of a large number of satellites once in space. Ideally, the programming of the satellite should be agile, but reliable allowing customization or improvements for the different batches. On the other hand, the current CubeSat missions operate few satellites, except for the Flock constellation from Planet Labs. However, this constellation does not have inter-satellite communication capabilities, which might allow the propagation of operational goals, such as monitoring a specific location. The flight software is the key system that might facilitate not only the propagation but also the automatic decomposition of the operational goals into each satellite tasks while avoiding commanding each satellite separately. In this context, a scalable flight software should facilitate the manufacturing (assembly, integration, and testing) and the operation of nanosatellite missions when the number of satellites in the constellation increases.

To identify these non-functional requirements in the following sections, we summarize the SUCHAI flight software non-functional requirements as follows:

- **Q1** The flight software must be **extensible**, in the sense that any change or improvement should be localized, avoiding affecting the system structure.
- **Q2** The flight software must be **modular**, so that non-critical modules, such as payloads, might be added or removed without affecting the entire system (e.g., needing to modify or recompile the entire system).
- **Q3** The flight software design should reduce failure points and help in the implementation of fault tolerance techniques to improve the mission **reliability**.
- **Q4** The flight software must be **portable** to multiple hardware and software platforms, such as embedded systems supported by FreeRTOS or computers capable of running GNU/Linux, being **reusable** along current and future missions.
- **Q5** The flight software must be **scalable**, in the sense that the solution can support the manufacturing and operation of an increasing number of nanosatellite in large constellations.

B. FUNCTIONAL REQUIREMENTS

To determine the functional requirements of the flight software, we considered the operation model (or use case) described in Fig. 1. When the satellite is within the range of the ground station, the operators can send telecommands to it. The satellite can execute these commands immediately (for example, download telemetry, download payload data or modify settings) or can queue commands in the flight plan for later execution (for example, sample sensors, take payload data or attitude control). During the rest of the orbit, the satellite has to perform some activities autonomously. These activities include periodic tasks (such as housekeeping, sending a beacon, resetting watchdog timers and waiting for incoming telecommands), execute commands queued in the flight plan at some specific time, and react to non-deterministic events such as malfunctions, unexpected resets, batteries discharge, among others. As a reference, a low orbit nanosatellite can establish contact with the ground

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

station three or four times in 24 hours and each contact may last between 5 to 10 minutes. The rest of the time, the satellite executes autonomous or scheduled operations.

This operation model is heavily based on the satellite's ability to execute commands, both remote or self-generated [18], [25]. The satellite operators should be able to break down the satellite mission in a series of commands. Autonomous operations are translated to self-generated commands with a well-defined execution logic (for example, periodical or event-based).

The concept of remote and self-generated command execution can also be extended to constellation operations. Let suppose that each satellite in the constellation can execute a set of commands. Mission control can decompose (manually or automatically) the mission goals into a series of commands that each satellite in the constellation has to execute. If we add inter-satellite communication capabilities, then any satellite can decide either to execute commands or delegate commands to surrounding satellites, thus facilitating the operation by distributing the mission goals over the constellation.

Therefore, the flight software functional requirements can be expressed in three generic enough sentences:

- **F1** The flight software must execute remote (on-demand) commands generated from ground satellite operators
- **F2** The flight software must execute self-generated (autonomous) commands, the execution logic of these commands can be single event, periodical, or event-based.
- **F3** The flight software must store and download telemetry data.

Note that any specific mission functional requirements should be translated to commands and command execution logic. Thus, the flight software is flexible enough to execute the requirement. For example: "to send a beacon once a minute", "to reset the satellite on demand" or "to collect particles-counter samples over the South Atlantic Anomaly", are all examples of possible mission functional requirements that can be implemented as commands executed with a defined logic.

III. FLIGHT SOFTWARE ARCHITECTURE DESIGN AND IMPLEMENTATION

A. GENERAL DESIGN

Following the experiences of similar projects [22], [23] the proposed software design follows the layer architectural pattern dividing the system in hardware drivers, operating system and application layers. This design provides a portable solution [44]–[46] that satisfies the requirement **Q4** because the operating system and the device drivers layer can be exchanged by design. This approach allow us to integrate existing solutions in the drivers and operating systems layers and focus in the design and implementation of an application layer that satisfies the operation model discussed in Section II-B.

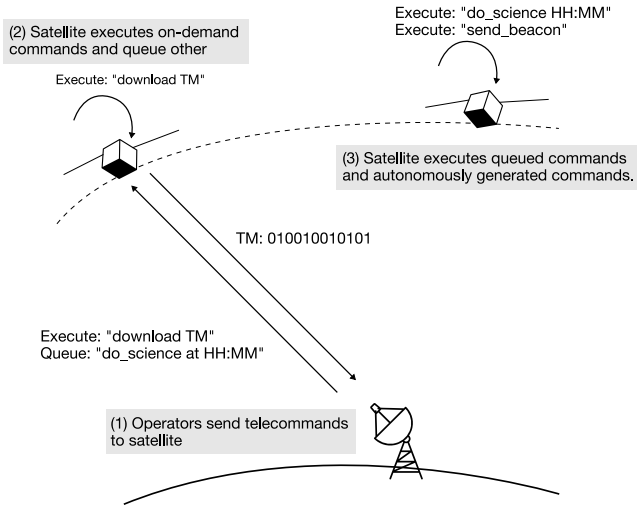

Figure 1. Example of satellite operations. Operators can send commands to the satellite which are executed immediately or queued in the flight plan. Additionally the satellite has to perform autonomous activities such as sending periodic beacons and executing the commands listed in the flight plan. All this activities may be considered commands, despite of they are remote or autonomous instructions.

We divided each layer functionality in modules and defined the possible dependencies between modules. Starting from the lower abstraction level, the minimum set of functionalities we required are: real time clock, data storage system, access to input/output devices and drivers for external devices or peripherals. The operation system layer will use these functionalities to provide high level features such as: threads, thread/task scheduler, timing functionalities, queues or message systems and synchronization structures. Finally in the application layer we require functionalities to implement tasks, data repositories, an implementation for the concept of commands and a main or application entry point. The dependency tree of these modules is described in Fig. 2

Maintaining this dependency tree helps to maintain portability because by design the operating system and drivers are totally independent from the application code. This architecture does not have any cycle between modules and between layers. Avoiding circular dependency is known to be effective on improving the maintenance and the comprehension of the overall architecture.

The following sections describe the design and implementation details of each layer, with special focus on the application layer implementation.

B. DRIVERS LAYER

This layer is populated by hardware or vendors dependent software, created to interact with peripherals and devices at a low level. Any supported device should provide a set of drivers, libraries or frameworks that help to interact with the device features. In our experience working with embedded systems, the diversity in this layer is so extended that we recommend following each vendor standard and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

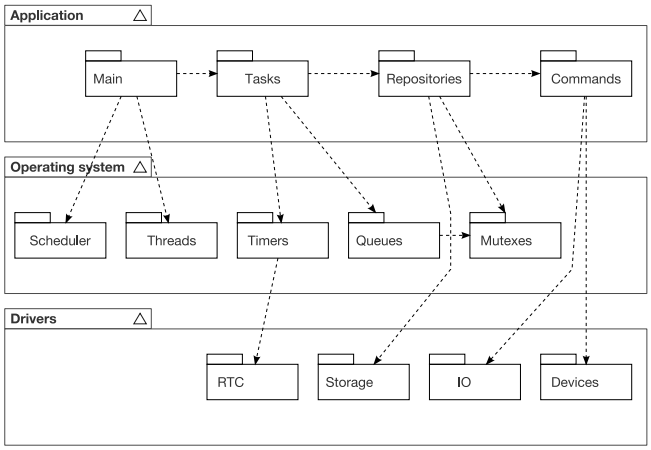

Figure 2. SUCHAI Flight software architecture: UML model diagram. Each layer consists of number of coarse-grain modules, a module being the result of compiling a number of C files and headers. A direct dependency between modules is indicated with an arrow. Our architecture follows a top-down interaction: higher level layers can interact with layers below, but a lower level layer should never depend on layers above.

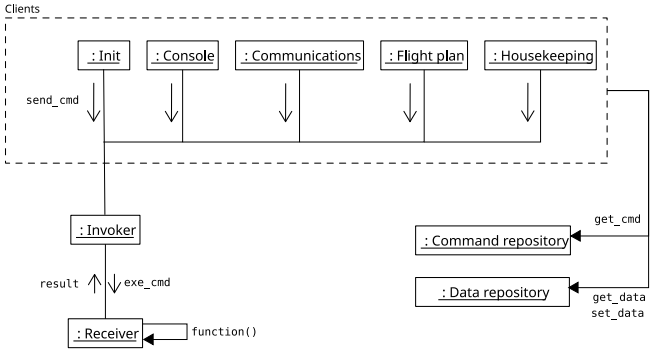

Figure 3. SUCHAI Flight software architecture: UML communication diagram. In this architecture clients only generate requests to execute commands, depending on the control strategy that each client implements. These requests are sent as messages to the invoker that may implement some control strategies over the command execution such as filtering, priorities, logging, among others. If the invoker decides that the command can be executed, it sends the request to the receiver. The receiver actually executes the command by calling the corresponding function. The command and data repositories provide an interface to handle commands creation and data storage respectively. From this diagram, we can extract the application messages path and how modules interact with each other.

solve the differences in the upper layer trough interfaces and wrappers. This recommendation includes managing different build systems at this level.

C. OPERATING SYSTEM LAYER

The Operating System (OS) adds an abstraction level between the hardware and application layers so more advanced solutions can be implemented in the application layer using utilities such as: multi-tasking, message queues, timers, files, among others. From requirement **Q4** and **Q5** we are interested in supporting at least two operating systems: GNU/Linux and FreeRTOS. Supporting GNU/Linux is useful for simulating the satellite functions in personal computers (a developer's laptop or testing servers) and to support powerful embedded computers such as the Raspberry Pi or the ARM™ Cortex A9 found in the Zynq 7000 family. Meanwhile, FreeRTOS is more suitable for low-power embedded systems, which are usually 16 or 32-bit microcontrollers such as the Microchip™ PIC24 and PIC32, the Atmel™ AVR32, the Espressif™ ESP32, to name a few. This portability layer is required to map specific operating system functionalities to our custom common interface. For example, we create our custom function `osTaskCreate()` to create Tasks, which is a wrapper to `pthread_create()` in GNU/Linux and to `xTaskCreate()` in FreeRTOS.

D. APPLICATION LAYER

Our solution is an application layer architecture based on the command processor design pattern. This pattern explains how to build an application that separates the service request from its execution, encapsulating each requirement in different commands [45]. However, this pattern was used at an architectural level and adapted for implementation in the C programming language [46]. The software architecture is described by the UML communication diagram shown in Fig. 3 and the UML sequence diagram shown in Fig. 4. The execution logic is the following: when a client is required to do a particular action, it creates a specific command and requests its execution to the invoker by sending the "`send_cmd`" message. The invoker checks if the command is executable and sends the requirement to the receiver as the "`exe_cmd`" message. The receiver actually executes the command by calling the corresponding function and sends the return value back to the invoker in a "`result`" message. Furthermore, the satellite needs to store at least the set of available commands, a list of settings or status variables, and data generated by payloads. Therefore, we included a set of repositories in the architecture, which are modules designed to encapsulate the data handling.

The architecture's necessary modules are the clients, the invoker and the receiver because they implement the command execution logic. As described in Listing 1, these modules are implemented as concurrent tasks (FreeRTOS) or threads (GNU/Linux) that use a messaging system to communicate clients' requests. These requests or commands are data structures (C structs) that contain all the relevant information to execute the target code such as a function pointer and parameters. Any of the many existing clients can generate commands. The messaging mechanism might be a shared queue where clients can push the commands as C structs; thus, the invoker can pop commands to be processed one at a time.

The SUCHAI flight software presents a clear execution logic because clients just generate the requirements and only the receiver actually calls functions for its execution; if additional control is required (such as command priorities, safe mode, execution logging, etc.), the invoker can implement these functionalities. This execution logic (described in Fig. 3 and 4) requires the definition of the set of client

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

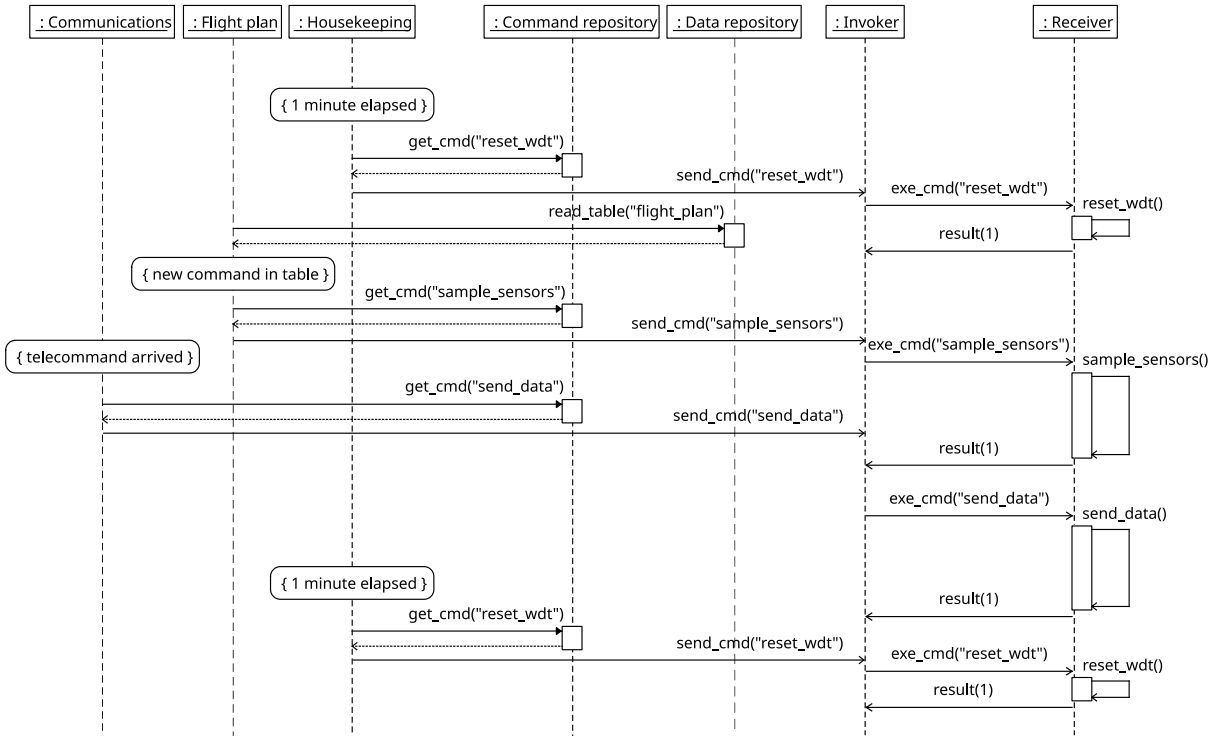

Figure 4. SUCHAI Flight software architecture: UML sequence diagram. Each client implements a control strategy and can request commands execution under certain circumstances. To execute a command the client has to create it using the command repository and then send an asynchronous message to the invoker. The invoker receives all client messages and organizes the execution by sending the request to the receiver. The receiver actually executes the command by calling the corresponding function. Once the command is executed, the receiver send a message back to the invoker with the execution result. From this diagram we can extract the application concurrency and the sequence of operations required to execute a command. We can see that clients works in parallel, that the invoker acts as a load balancer or proxy for the requests, and that the receiver performs the heavy work executing the code corresponding to each command. Note that clients send execution requests asynchronously and do not have direct feedback of the execution result.

modules, commands and repositories.

1) Client modules

Each client module controls a specific subsystem such as a device (radio, EPS) or a payload (camera, sensors) by requesting commands for execution. Each client may implement particular control strategies, such as listening for events, incoming telecommands or periodic timers. For example, Listing 1 shows that to reset the internal watchdog timer the "`reset_wdt`" command is generated at intervals of one second.

Note that clients can be modified, added or removed without affecting other clients nor the command execution mechanism. The architecture supports several clients working concurrently, which expresses a high level of modularity as formalized in requirement **Q2**. The final set of client modules depends on the specific mission requirements. As shown in Fig. 3, the SUCHAI flight software includes the following set of default client modules that meet the requirements of this series of nanosatellites:

- **Initialization**: Executes initialization activities such as subsystem configurations, starts other clients depending on the operational mode (safe mode, normal mode, science mode), post-deployment silent time, initial detumbling, among others.
- **Debug console**: This module includes functionalities to execute commands on-demand using a serial or remote terminal. It is essential for debugging during the development and pre-launch stages, although it is not useful once in orbit.
- **Communications**: Receives telecommands from the ground station and parses these telecommands to generate the corresponding system commands.
- **Flight Plan**: Schedules commands to be executed at a certain date and time. The schedule can be modified by specific commands to dynamically change the mission plan once the satellite is in orbit.
- **Housekeeping**: All the activities related to control the satellite status and health are included here. Most of them require the generation of commands at specific time intervals.
- **Payloads**: The coordination and control logic of payloads is isolated in a dedicated client. This client module implements the requirements of each payload.
- **Watchdog**: Periodically resets both internal and external watchdog timers, also implements a software watchdog in case no telecommands are received during a certain time.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

Listing 1. Application layer implementation example

```
/* Example Client thread */
void client(void){
  while(1){
    /* execute every 1 second */
    sleep(1000);
    /* create a command */
    cmd_t *send_cmd = cmd_get("reset_wdt");
    /* send command as a message to invoker */
    send_message(invoker_queue, send_cmd);
  }
}

/* Invoker thread */
void invoker(void){
  while(1){
    /* read a command sent by a client */
    cmd_t *exe_cmd = receive_message(
        invoker_queue);
    /* check if the command is executable */
    if(check_if_executable(exe_cmd)){
      /* send the command to the receiver */
      send_message(receiver_cmd_queue, exe_cmd
          );
      /* wait and receive the command result*/
      int result = receive_message(
          receiver_stat_queue);
      /* keep an execution log */
      save_execution_result(exe_cmd, result);
    }
  }
}

/* Receiver thread */
void receiver(void){
  while(1){
    /* wait and receive command from invoker
        */
    cmd_t *run_cmd = receive_message(
        receiver_cmd_queue);
    /* execute the command */
    int result = run_cmd.function(
                    run_cmd->fmt,
                    run_cmd->params,
                    run_cmd->nparams);
    /* Send the result back to the invoker */
    send_message(receiver_stat_queue, result);
  }
}
```

Listing 2. Command implementation example

```
/* Command interface definition */
typedef int (*cmd_function)(char *fmt, char *
    params, int nparams);

/** Register functions in command repository
 **/
int cmd_trx_init(void){
  cmd_add("update_beacon", cmd_update_beacon,
      "%d %s", 2);
}

/** Updates the beacon content and period.
 **/
int cmd_update_beacon(char* fmt, char* params,
     int nparams){
  int period;
  char beacon[10]; /* Max 10 characters  */

  if(sscanf(params, fmt, &period, &becon) ==
      nparams) {
    printf("Parsed: %d, %s", n, s);
    trx_set_beacon_period(period);
    trx_set_beacon_text(beacon);
    return 1;
  } else{
    printf("Failed parsing parameters"):
    return 0;
  }
}
```

Listing 3. Client implementation example

```
#include "repoCommand.h"

void taskHousekeeping(void *param){
   int elapsed = 0;
   while(1){
     _sleep(1000); // Sleep 1 second
     elapsed ++;   // Seconds counter
     // Execute every 10 minutes
     if((elapsed % 60*10) == 0){
       // Get command by name
       cmd_t *new_cmd = cmd_get_str(
                          "update_beacon");
       // Set command parameters
       cmd_add_params(new_cmd, 3, "SUCHAI");
       // Send command to execution
       cmd_send(new_cmd);
     }
   }
}
```

2) Commands

Commands are implemented as functions that share a common interface. As shown in Listing 2 we defined an interface in C (a typedef) that all commands must implement. Thus, the receiver can make a generic function call. Any command is identified by an unique pair of name and numeric ID. They can receive an arbitrary number of parameters, which can be either numeric or text data types. The implementation of a command is very specific, but usually consists of a wrapper to low-level functions such as device-driver calls, communication with external subsystems or read/write processes. For example, Listing 2 shows the implementation of the command that sets the text and period of the satellite's beacon. The command receives two parameters, the period in minutes and the beacon text as a string, and then calls the specific driver functions to configure the satellite transceiver.

The implementation of this command is independent from the rest of the system and its functionality may be tested separately.

Commands are implemented in separated files and grouped by functionalities. The `cmd_update_beacon` is related with the transceiver functionalities, so it is implemented in the `cmdTRX.c` and `cmdTRX.h` files. Then, this command has to be registered in the command repository to be available in the system. We use the `cmd_add` function, part of the command repository API available in `repoCommand.h`, to register the command name, the function, the parameters format and the number of parameters.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

Once implemented and registered, commands can be searched and executed using the command repository API. Usually in a client module (example `taskHousekeeping.c`) we include `repoCommand.h` to get commands, fill parameters and request command execution. As shown in Listing 3, the `taskHousekeeping` client requires updating the satellite beacon every ten minutes. Consequently, the satellite creates a new "`update_beacon`" command with the corresponding parameters and sends the command for execution every ten minutes.

3) Repositories

Repositories are modules that provide a uniform and thread safe methods to access (read, write) all data in the system and to organize the available storage space. In this design, three data types were identified: the system status variables, the list of available system commands and the payloads data. To handle these data types and their attributes, the following repositories are defined:

- **Command repository**: This repository is used to store and give access to the commands available in the system. Clients use this repository to create a new command with specific parameters. This repository is initialized at boot up time and provides read-only access.
- **Status repository:** This repository provides access to the system status variables such as operational parameters (battery voltage, temperatures, date and time, position, available energy level, among others), system health reports (*e.g.,* reset counter, last reset source, number of days since the last connection with ground station) and system settings (including operating mode, enabled subsystems, communication baud rates and beacon periodicity). Some of these variables require a persistent storage to maintain the system coherence and configuration in case of unexpected resets. In GNU/Linux a SQL database is used, while in the SUCHAI 2 and 3 OBC (GomSpace™ Nanomind A3200) these variables are stored in an external FRAM chip.
- **Data repository:** General data such as payload results, system logging, flight plan schedule or telemetry waiting to be downloaded is accessed using this repository. This repository requires a massive and persistent storage system such as and external SD memory, and the concurrent access needs to be synchronized to avoid data races.

The SUCHAI flight software is a free and open-source (FOSS) project licensed under the GPLv3 license. The source is hosted on GitHub and can be found in the following link: https://github.com/spel-uchile/SUCHAI-Flight-Software. Installation and execution instructions are available in the repository and can be tested on any computer running GNU/Linux.

IV. VALIDATION

As discussed in SectionII, the CubeSat community is concerned about the design and quality of the flight software. However, defining and ensuring rigorous software quality criteria is not a simple task. CubeSat projects apply different techniques such as extensive testing [16], [36], hardware in the loop simulation [37], static analysis, or the use of certified language standards [9]. Our approach utilizes software engineering tools, in the context of embedded systems development, to track the quality attributes of the flight software using a visual architecture evaluation tool. If this visual tool is integrated with the development process, we can monitor the evolution of the non-functional requirements and early detect architecture disruptions that may deteriorate the software quality.

Visual displays allow the human brain to study multiple aspects of a complex problem in parallel. It is well known that software visualization allows for a higher level of abstraction and a closer mapping to the problem domain [47]. For this reason, we have produced several visualizations to measure and assess the modularity of the components involved in our flight software and to extract the architecture from the source code. Source code visualizations are generated using a script written in the Pharo programming language [48] and based on an agile visualization library called Roassal [49]. Instructions to use the visualization script are available in the Git repository server: https://github.com/spel-uchile/SUCHAI-Flight-Software/blob/master/test/viz/dependency_graph.md. Additionally we created a simple web interface, available on https://data.spel.cl/ to interactively navigate across this visualization timeline.

Our validation methodology is complemented with automatic cross-compilation and automated testing to evaluate portability and reliability of the software. The tools presented in this section have been included in a continuous integration server built with Jenkins. On every commit a set of validations scripts are executed in a pipeline and the results are reported in the web platform: https://jenkins.spel.cl/. As Fig. 5 shows, the stages included in the pipeline are:

1) Fetch the SUCHAI Flight Software source code from GitHub.
2) Generate a set of visualizations of the source code architecture.
3) Compile the code for GNU/Linux.
4) Recompile the code for three embedded architectures (AVR32 UC3, ESP32, and Nanomind A3200)
5) Run the test suite in GNU/Linux.

The continuous integration system helps us ensure that changes in the source code do not break the compilation in any of the supported platforms or the expected behavior of the software. Furthermore, extracting visualizations in each commit tracks changes in the architecture and allows developers to take actions when the commit is made avoiding long divergences along the project history. In the same way, automated testing increases software reliability by adding traceability to encountered errors.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

Figure 5. Capture of the Jenkins pipeline status. The continuous integration system can run several stages to extract visualizations, build, and test the source code to detect any error introduced in a commit.

A. EVALUATION OF MODULARITY USING SOFTWARE VISUALIZATION

Visualizing software dependencies is a common technique employed to communicate interaction between components [50]. In the global architecture, these interactions express the modularity of the flight software solution. Our visualization tool consists of a script that parses the files in the source code, classifying them in application (including main, clients, invoker, receiver, repositories and commands files), operating system and drivers and associate a color to each of these components. Then, constructs a directed graph based in its dependencies. Dependencies are extracted from the `#include` directives contained in the source code. Edges between modules indicate a dependency as described in the UML model diagram shown in Fig. 2. Two snapshots of the SUCHAI flight software source code are represented in Fig. 6. The first snapshot was produced in December 2017 (commit `765c128` on GitHub) while the second in May 2018 (commit `0ca21db` on GitHub).

Our visualization shows modules (files having the extension `.c` with the corresponding `.h`) and their dependencies. Each file is represented as a colored box. The height of the box indicates the number of lines of code contained in the module while the width of a box represents the number dependencies included in the represented module. For example, the central green box in Fig. 6 represents the module named `repoCommand.c`, which represent the command repository whose purpose is to store and give access to available commands. This module is one of the largest modules of the SUCHAI flight software since it is the highest box. Similarly, the blue box on the top represents the `main.c` file. The `main` is the widest module because it includes a large number of dependencies, which makes sense because it is the software entry point.

In the commit `765c128` from December 2017 the flight software contains only the fundamentals modules to support command executions. The diagram in Fig. 6 left shows that client tasks depend on command and data repositories, command repository includes all command modules, and data repository includes drivers for data storage handling. This dependencies graph matches the proposed architecture described in Fig. 2, except for a circular dependency between the command repository and the command modules. This circular dependency is not described nor desired in the architecture shown in Fig. 2. However, inspection to the source code reveals that using a command repository function inside command modules to register new commands in the system increases the readability and maintainability of the code.

The diagram of the commit `0ca21db` from May 2018 in Fig. 6 right shows the evolution of the software after several commits. In a similar analysis, we determined that the architecture is preserved and that no extra dependencies were added. However, we observe that new clients were added, one was deleted, new commands were added, while some modules changed their amount of code. The main module has reduced the lines of code since some initialization routines were moved to the new `taskInit` client. On the one hand, new commands and clients were added, which means new functionalities; but on the other hand, the `Invoker` and the `Receiver` remains intact, which means that the commands execution logic was not intervened. A significant number of lines of code have been added to the data and command repository; hence, we should concentrate on testing these modules.

These visualizations can be immediately exploitable by a software engineer. They are meant to be an early indicator of (i) a violation of the architecture and (ii) an anomaly that may be due to exceptional entities. This case can be observed in Fig. 7 that corresponds to the commit `f1d695a` from December 2017. The visualization reveals that a new client module - the flight plan - was added. Additionally, we can see a dependency between the `taskFlightPlan` client module and the `data_storage` driver. Although this commit produced a functional code that passed all the tests, this new dependency disrupts the proposed architecture, because a client task should not directly execute code of a driver module as described in Fig. 2. The `taskFlightPlan` client should use the data repository API or a command instead.

Another example is the SUCHAI I flight software, which is already in orbit and have been working properly for more than a year. This software also used this architecture but Fig. 8 shows a complex dependencies tree between modules. This is a symptom of high software complexity where several disruptions of the architectural rules had a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

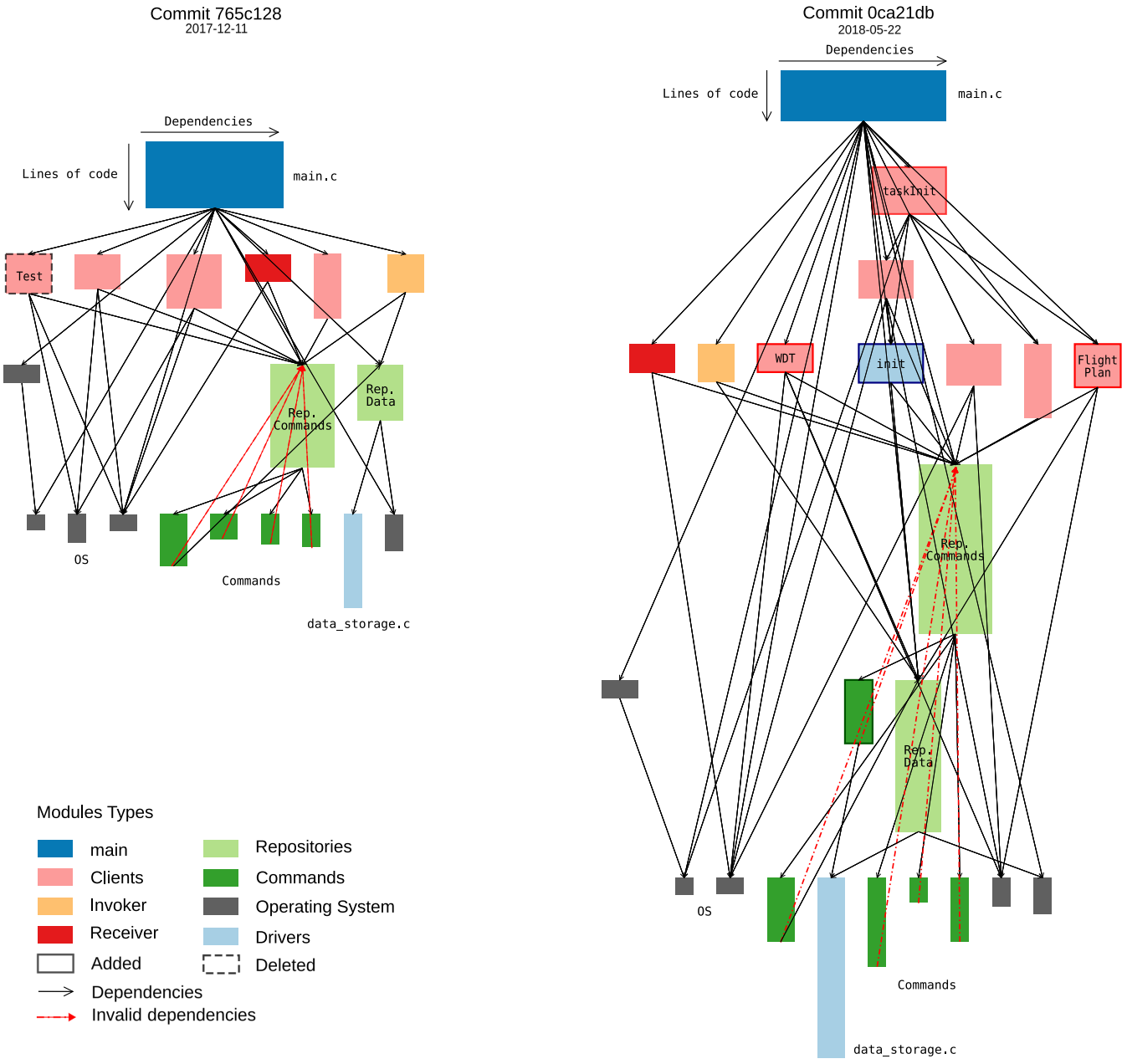

Figure 6. Modules dependencies comparison between commits `765c128` and `0ca21db`.

negative impact on the quality attributes.

B. EVALUATING APPLICATION LAYER ARCHITECTURE USING SOFTWARE VISUALIZATION

The same visualization tool is used to validate the architectural rules of the application layer. The application layer architecture is based on the command pattern as the UML diagram in Fig. 3 details. Modules in the application layer are related by a messaging system to send commands from Task modules to the Invoker and the Receiver. The messaging system was implemented as queues in FreeRTOS and Linux, so we are interested in visualizing the relation between Task modules and queues. Figure 9 shows the structure and evolution of the application layer after several months of development. Only Task modules of type Client are using the queue to send commands for execution. Although some Client modules were added and removed from commit `765c128` to `0ca21db`, the architecture of the application layer remains intact. The evolution shown in Fig. 6 contrasts from the results in Fig. 9 because with time, new features, lines of code and modules were added, but the execution logic in the upper layer has remained intact.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

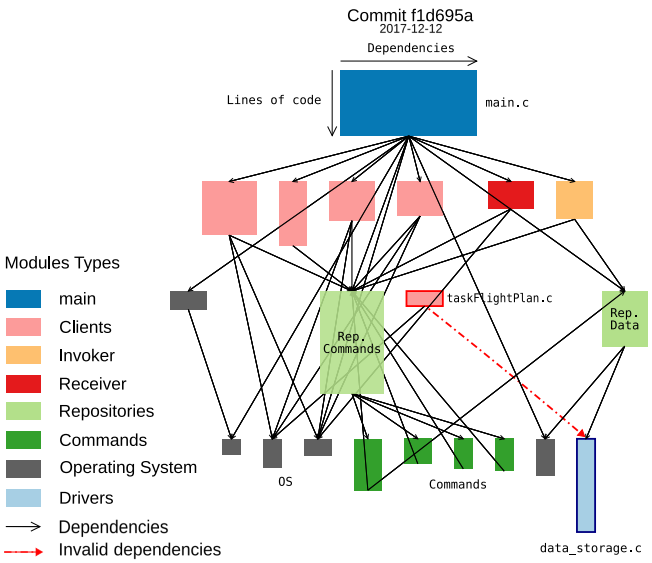

Figure 7. Example of an architecture disruption in commit `f1d695a`. The flight plan module includes code from the data storage driver, but this is not allowed in the architecture. Client modules should use the data repository API instead.

C. EVALUATING RELIABILITY AND PORTABILITY USING A CONTINUOUS INTEGRATION SYSTEM FOR TESTING

1) Testing software reliability

Our strategy to ensure software reliability is based on automated testing, which includes unitary and integration tests. Unitary tests can be applied to individual commands, because they are functions that can be called independently, as an API or simple interfaces. A command developer should provide the corresponding unitary test, and Jenkins will execute automatically the unitary tests using CUnit framework.

Integration tests are mini-applications that run full use cases such as running a command with different variables, running load test by executing hundreds of commands in a second, adding and removing commands in the flight plan, among others. We expect that the flight software can run these cases without crashing, finishing within a certain time, and that all commands return a valid code. These behaviors are tested by comparing execution log outputs with the expected ones; if they do not match, the maintainers can review the report and fix potential issues.

2) Testing software portability

The three-layer architecture used to design and program our flight software should allow porting it to new platforms straightforward. As described in Section III, the OS layer has been ported to GNU/Linux and FreeRTOS. Running the software in GNU/Linux helps us to test and debug the fundamental logic of commands execution as well as to test any change in the API or interfaces. For new collaborators, especially non-embedded systems developers, it is significantly easier to program new features in their laptops running GNU/Linux rather than working with an embedded system such as the Atmel™ AVR32 included in the Nanomind A3200, which is actually the flight version and cannot be damaged in the developing process. Later, the results obtained in a GNU/Linux platform have to be replicated in the satellite OBC with the help of more experienced team members. Section II described how many CubeSats are using GNU/Linux in their flight software.

FreeRTOS is an operating system that supports many of the embedded platform currently used in CubeSats such as the PIC24F (SUCHAI 1), the AVR32 (Nanomind A3200 used in SUCHAI 2 & 3), among others. Consequently, we ported the drivers' layer to three different hardware platforms using the SDK available from manufactures:

- **Atmel™ AVR32UC3**: we use the UC3-A3 XPLAINED evaluation board, which mount an AT32UC3A3256 microcontroller, as a low-cost alternative to the Nanomind A3200. In this platform, we can test the majority of the SUCHAI 2 and 3 flight software features without compromising the flight-model OBC. We use the Atmel™ ASF libraries, which include drivers, utilities, and FreeRTOS v7.0.0 for this particular board.
- **Espressif™ ESP32:** this is a popular, low-cost, embedded system that integrates a two-core processor with WiFi and Bluetooth capabilities. Because this platform supports FreeRTOS, using the SUCHAI flight software here may extend the usage of our solution to the Internet of Things (IoT) applications. Espressif™ provides an SDK with drivers, libraries and FreeRTOS v8.2.0.
- **Gomspace™ Nanomind A3200:** this is the actual OBC used in the SUCHAI 2 and 3 CubeSats. It mounts an AT32UC3C0512C microcontroller and Gomspace™ provides a customized version of the Atmel™ ASF, which includes drivers, utilities and FreeRTOS v8.0.0 for this particular OBC.

Through a simple configuration file (`config.h`) we can select the target platform and run the corresponding compilation script. Listing 4 shows a configuration file ready to compile for the Nanomind platform; if we comment the `FREERTOS` define in line 3 and uncomment `LINUX` in line 2, the software can be compiled for GNU/Linux. If a command has to implement code for a specific platform, it is possible to use this definition to do a conditional compilation.

Our pipeline in Jenkins automatically changes this configuration file, compiles for every platform, and generates the binaries. If a change breaks the code for a specific platform, the maintainers are notified, so they can react and fix any potential problem.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

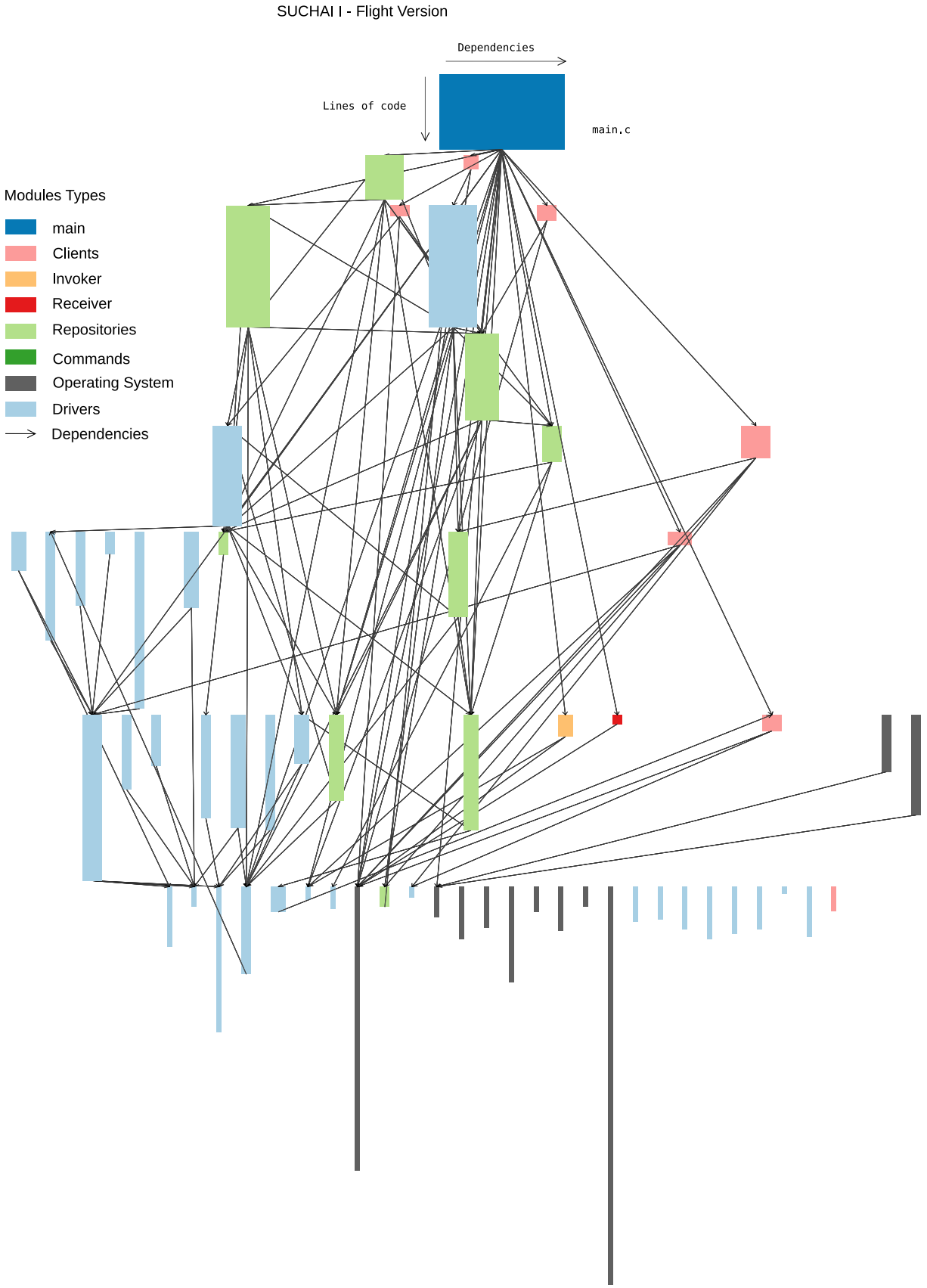

Figure 8. Modules dependencies visualization of SUCHAI I flight software. This flight software is currently in orbit.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

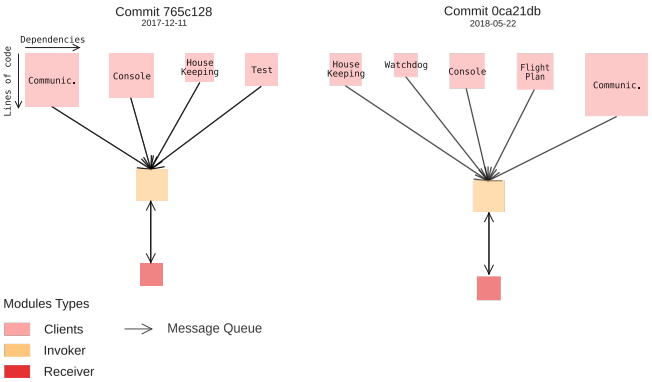

Figure 9. **Application layer architecture visualization. Relation between Task modules, Invoker, Recevier and messages queues for commits `765c128` and `0ca21db`.**

Listing 4. **Configuration file**

```
/* Select one operating system */
// #define LINUX     // Use Linux
#define FREERTOS // Use FreeRTOS (select arch)
/* Select the correct architecture */
#ifdef FREERTOS
// #define ESP32    // Run in ESP32
// #define AVR32    // Run in AVR32
  #define NANOMIND // Run in Nanomind A3200
#endif

/* System debug configurations */
// Debug levels
#define LOG_LEVEL LOG_LVL_INFO
...
```

Listing 5. **Adding commands to existing modules**

```
//File: cmdObc.h
#include "wdt.h"
/** Register on OBC commands
 */
void cmd_obc_init(void);

/** Reset the watchdog timer
 */
int obc_reset_wdt(char *fmt, char *params, int
      nparams);

//File: cmdObc.c
void cmd_obc_init(void){
   cmd_add("reset_wdt", obc_reset_wdt, "", 0);
}

int obc_reset_wdt(char *fmt, char *params, int
      nparams){
   return = wdt_reset();
}
```

V. CASE STUDY: EXTENDING THE SOFTWARE

In this section, we aim to exhibit attributes such as extensibility and modularity as described in requirements **Q1** and **Q2** by extending the software functionalities. The SUCHAI flight software has two strategies to extend the system functionalities. The first one is the addition of new commands that can be executed from any existing client.

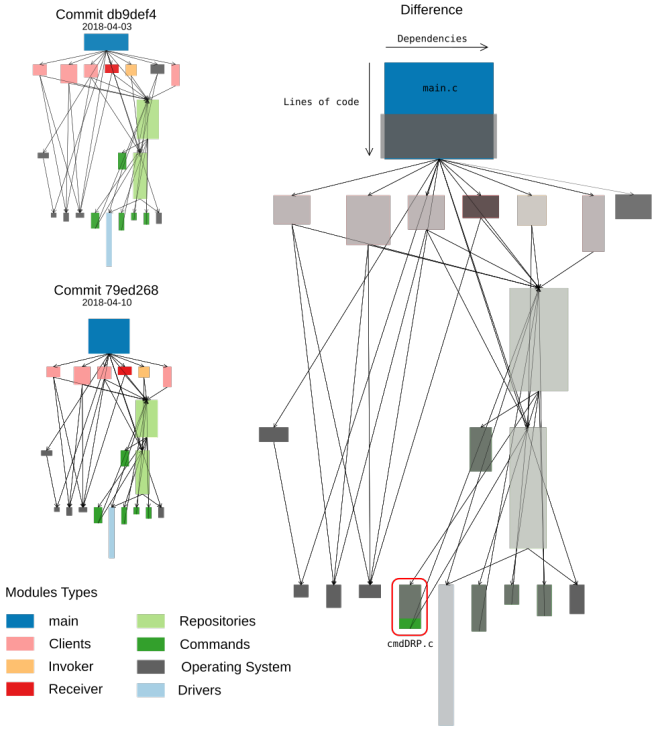

Figure 10. **Comparing the software architecture after adding a command in commit `79ed268`. We superimpose both diagrams and compare changes using colors differences. Gray means that both diagrams are identical. However, differences are detected in the main module and in the data repository commands module (`cmdDRP.c`)**

For example, if we implement the *"reset watchdog timer"* command, it can be issued by a remote telecommand or the satellite can execute it periodically. The second extension strategy is the addition of a new client that can execute any of the existing commands to achieve a specific control goal. For example, to add a new client that implements a *"software watchdog timer"* which resets the satellite if a certain telecommand has not been sent during a specific period.

A. EXTENDING BY ADDING NEW COMMANDS

1) Adding commands to existing modules

Let consider that the *"reset watchdog timer"* command is related to the commands that manage the OBC functionalities, so we included it in the existing `cmdOBC.c` file. Also, that the OBC vendor provides a driver module called `wtd.c/.h` to use the watchdog timer peripheral. As shown in listing 5, **only five lines of code** (excluding comments) were necessary to implement and register a new command called "`reset_wdt`". Our visualization tool in Fig. 10 shows that this modification **only affects one module** keeping the application logic intact. Although the command has been implemented and registered, it has not being used by any client module yet, so adding this new feature has little or no impact in the entire system.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

2) Adding a new commands module

In contrast, if we consider that the watchdog timers related commands should be part of a separated module, we can create a new pair of files called `cmdWDT.c` and `cmdWDT.h` and implement the reset-watchdog command inside. As described in listing 6 we **added 9 lines of code in four files, one new module was added, and the `repoCommand.c` module was modified**. This is because command modules need to be registered in the command repository, which is done by calling an initialization function. The `cmd_repo_init()` function in the command repository is called in the `main` function to initialize all command modules.

B. EXTENDING BY ADDING NEW CLIENTS

In this section, we implement the control of two watchdog timers using the commands implemented previously in Section V. Thus, the flight software was extended by adding a new client module to perform the control logic described in the Algorithm 1. In this logic, the client performs two tasks. First, periodically reset the OBC watchdog timer (as a signal of correct functioning) using the "`reset_wdt`" command. Second, implements a *"software watchdog timer"* by sending the "`reset`" command if the variable "`elapsed_gnd_timer`" has not been cleared by a telecommand (which is a signal of system malfunction).

This client was implemented in the `taskWDT.c` and `taskWDT.h` files; additionally, the task is launched from the `main` function as described in Listing 7. In this case, the modification consists of the implementation of simple control logic to send commands periodically. To use the commands, we included the command repository API (`repoCommand.h`). Similarly, the data repository API from `repoData.h` was included to read and update system-wide variables such as the timer for the software watchdog. This variable should be reset by a command executed from the ground station, or the WDT client will send the "reset" command. Figure 11 was generated using our visualization tool and showed that only the main module was modified, and one new module was created. No dependencies with existing modules were added except by the usage of the command and data repository APIs.

Algorithm 1 Control logic to reset watchdog timers

```
max_obc_wdt ← 10
max_gnd_wdt ← 3600 * 48
elapsed_obc_timer ← 0
elapsed_gnd_timer ← 0
loop
  sleep 1 second
  elapsed_obc_timer ← elapsed_obc_timer +1
  elapsed_gnd_timer ← elapsed_gnd_timer +1
  if elapsed_obc_timer > max_obc_wdt then
    elapsed_obc_timer ← 0
    send command "reset_wdt"
  end if
  if elapsed_gnd_timer > max_gnd_wdt then
    send command "reset"
  end if
end loop
```

Listing 6. Adding commands to existing modules

```
1  //File: cmdWDT.h
2  #include "wdt.h"
3  /**Register on watchdog timer (WDT) commands
4   */
5  void cmd_wdt_init(void);
6
7  /** Reset the watchdog timer
8   */
9  int wdt_reset_timer(char *fmt, char *params,
       int nparams);
10
11 //File: cmdWDT.c
12 void cmd_wdt_init(void){
13    cmd_add("reset_wdt", wdt_reset_timer, "",0)
          ;
14 }
15
16 int wdt_reset_timer(char *fmt, char *params,
       int nparams){
17    return wdt_reset();
18 }
19
20 //File repoCommand.h
21 #include cmdWDT.h
22
23 //File repoCommand.c
24 int cmd_repo_init(void){
25   // Init existing repos.
26   cmd_test_init();
27   cmd_obc_init();
28   cmd_drp_init();
29   // Init new cmd repo.
30   cmd_wdt_init();
31
32   return CMD_OK;
33 }
```

Listing 7. Adding new client module

```
1  //taskWDT.h
2  #include "repoCommand.h"
3  #include "repoData.h"
4
5  void taskWDT(void *param);
6
7  //taskWDT.c
8  #include "taskWDT.h"
9
10 void taskWDT(void *param) {
11
12    // Seconds to send "reset_wdt" command
13    unsigned int max_obc_wdt = 10;
14    // Seconds to send "reset" command (48hrs)
15    unsigned int max_gnd_wdt = 3600*48;
16    // OBC timer counter
17    unsigned int elapsed_obc_timer = 0;
18    // Get GND timer counter
19    unsigned int elapsed_gnd_timer = 0;
20
21    while(1) {
22      // Sleep task to count seconds
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access


```
23        osDelay(delay_ms);
24        elapsed_obc_timer ++;
25        // Get current counter value
26        elapsed_gnd_timer = dat_get_system_var(
              dat_gnd_wdt);
27        // Update GND timer counter
28        dat_set_system_var(dat_gnd_wdt,
              elapsed_gnd_timer+1);
29
30        // Periodically reset the OBC watchdog
31        if(elapsed_obc_timer > max_obc_wdt) {
32          elapsed_obc_timer = 0;
33          cmd_t *rst_wdt=cmd_get_str("reset_wdt")
                ;
34          cmd_send(rst_wdt);
35        }
36
37        // If nobody reset elapsed_gnd_timer
38        // then reset the OBC
39        if(elapsed_gnd_timer > max_gnd_wdt) {
40          cmd_t *rst_obc = cmd_get_str("reset");
41          cmd_send(rst_obc);
42        }
43      }
44  }
45
46  //main.c
47  #include taskWDT.h
48
49  int main(void)
50  {
51
52    /* On reset */
53    on_reset();
54    /* Initializing shared Queues */
55    dispatcher_queue = osQueueCreate(25,
56                            sizeof(cmd_t *));
57    executer_stat_queue = osQueueCreate(1,
58                            sizeof(int));
59    executer_cmd_queue = osQueueCreate(1,
60                            sizeof(cmd_t *));
61
62    os_thread threads_id[7];
63
64    /* Crating system task (the others are
          created inside taskDeployment) */
65    osCreateTask(taskDispatcher,"invoker",
          15*256, NULL, 3, &threads_id[0]);
66    osCreateTask(taskExecuter, "receiver",
          15*256, NULL, 4, &threads_id[1]);
67
68    /* Creating clients tasks */
69    osCreateTask(taskWDT, "WDT", 15*256, NULL,
          2, &threads_id[2]);
70    osCreateTask(taskConsole, "console", 15*256,
           NULL, 2, &threads_id[3]);
71    osCreateTask(taskHousekeeping, "housekeeping
          ", 15*256, NULL, 2, &threads_id[4]);
72    osCreateTask(taskCommunications, "comm",
          15*256, NULL,2, &threads_id[5]);
73    osCreateTask(taskFlightPlan,"flightplan"
          ,15*256,NULL,2,&threads_id[6]);
74
75    /* Start the scheduler. Never return */
76    osScheduler(threads_id, n_threads);
77    return 0;
78  }
```

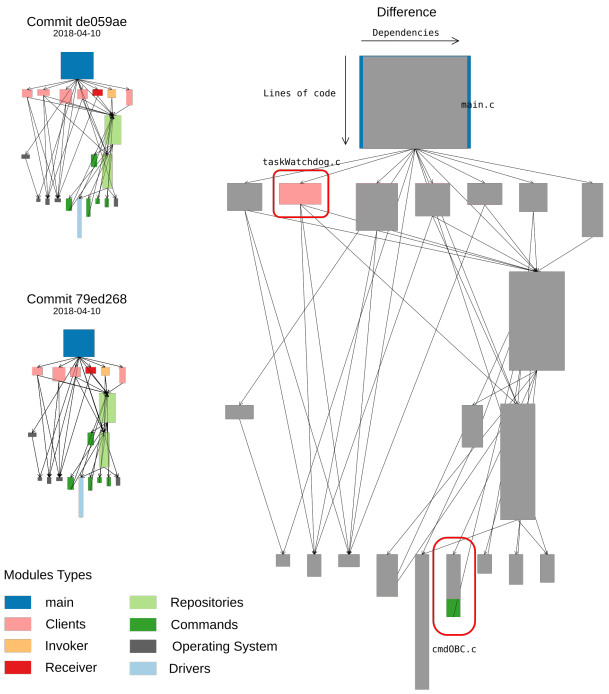

Figure 11. Comparing software architecture after adding a client module and new commands in commit `de059ae`. The module task watchdog (`taskWatchdog.c`) was added, and new lines of code were added to the OBC commands module (`cmdOBC.c`).

VI. DISCUSSION

A. THE SUCHAI CUBESATS PROGRAM EXPERIENCE

The development of this flight software started with the SUCHAI 1 CubeSat, launched into space in 2017. This first version was successfully proven in space, demonstrating that the concept of the command executor architecture is feasible and the functional requirement were met. However, due to common limitations in the development of CubeSat projects, we had concerns about the quality attributes and the coverage of non-functional requirements of the implemented software. When the visualization tools where first developed and used to analyze the SUCHAI 1 flight software we realized that, at some point of the development history, source code changes could disrupt the architectural rules without affecting functionalities (See Fig. 8. As a result, it was difficult to verify quality attributes or the software architecture itself.

With more CubeSat in the production line (SUCHAI 2, 3 and PlantSat), we decided to re-implement the same software architecture of SUCHAI I in a second version of the flight software, as described in Section III. In this occasion, we integrated the visualization tool, tests and cross-compilation tasks in a continuous integration server. The requirements and design guidelines were the same as the first CubeSat, but this time the software quality attributes have been tracked closely during the development process.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

The results of Section IV show that the automatically generated visualizations can be compared with the architecture UML diagrams shown in Figures 2 and 3 to determine architectural rules breaks, such as incorrect dependencies or miss-usage of the messages paths.

Up to this moment, the concept of a flight software based on the command design pattern has demonstrated being useful in the nanosatellite program of the University of Chile. Functional requirements related to periodical telemetry acquisition, localized telemetry acquisition, system integrity operations (housekeeping), and mission re-planning has been successfully translated to commands and implemented in these satellites. This solution has also demonstrated advantages in minimizing and debugging software errors, because, independently of the task, the software functioning is well known.

Using the portability capabilities, we have discovered advantages in using the same flight software in the ground segment (x86_64 computers with GNU/Linux) and also in related projects, such as balloons with radiosondes (based on Raspberry Pi).

B. VISUALIZATIONS UNDER TEST

To determine the usability of the visual code analysis tool presented in section IV, we conducted a user-study experiment with a group of 11 persons that we considered potential flight software developers in an educational CubeSat project. The group was composed of undergrad students (5), grad students (2), and engineers (4) with software development skills mainly in Python (10), C (8), Java (8), and C++ (6). In total, four (4) individuals have worked with the SUCHAI Flight Software previously, three (3) have used a visualization tool before, and the authors were not part of the experiment. Participants were asked to perform a series of tasks using the visualization to extract relevant information about the software architecture and the impact of code changes in global architecture. These tasks were divided into four steps, described as follows:

- **T1. Read a description of the visualization objectives, interact with a figure similar to Fig. 6 from commit `765c128` and answer the following questions:**
  - Q1. In a scale from 1 to 5. How much do you understand the visualization?
  - Q2. Which module contains the most lines of code?
  - Q3. Which module contains the most dependencies?
  - Q4. Which module depends on a driver?
- **T2. Understand the architecture using Fig. 2 and find architecture disruptions using Fig. 7:**
  - Q1. Can a "Task" module use a "Command" module directly in Fig. 2?
  - Q2. Can a "Task" module use the "Queue" module directly in Fig. 2?
  - Q3. Can a "Task" module use any "Driver" module directly in Fig. 2?
  - Q4. Does a "Task" module of type "Client", "Receiver" or "Invoker" use a "Command" module directly in Fig. 7.
  - Q5. Does a "Task" module of type "Client", "Receiver" or "Invoker" use the "Queue" module directly in Fig. 7.
  - Q6. Does a "Task" module of type "Client", "Receiver" or "Invoker" use any "Driver" module directly in Fig. 7.
- **T3. Determine changes in the architecture comparing the visualization of two commits in Fig. 6:**
  - Q1. How many modules were added in the Fig. 6 commit `0ca21db`?
  - Q2. Considering the modules existing in both commits of Fig. 6. Which modules were significantly changed?
  - Q3. Which module was deleted in Fig. 6 from commit `765c128` to commit `0ca21db`?
- **T4. Determine changes in the architecture comparing the visualization of two commits in Fig. 10**
  - Q1. Which modules present changes between commit `db9def4` and commit `79ed268` in Fig. 10?

Table 2. Summary of survey results

Task	Question	Answers		Correct answer
		Correct	Incorrect	
T1	Q1	NA	NA	11 subjects understood the visualization (option 5)
	Q2	9	2	repoCommand.c
	Q3	10	1	main.c
	Q4	10	1	repoData.c
T2	Q1	11	0	No
	Q2	11	0	Yes
	Q3	11	0	No
	Q4	3	8	Yes
	Q5	2	0	No
	Q6	5	6	Yes
T3	Q1	5	6	4
	Q2	9	2	main.c, repoData.c, repoCommand.c, datastorage.c, cmdOBC
	Q3	2	9	taskTest.c, cmdTestCommand.c
T4	Q1	6	5	main.c, cmdDRP.c

Table 2 summarizes our survey results. It shows that the proposed visualization tools can be useful visualizing how changes in the source code affect the general structure of the software. If considerable changes in the number of lines of code or dependencies are detected in some modules, then it is necessary to check that code in detail. Architectural rules issues, such as incorrect dependencies can be detected with a detailed analysis of the figures. However, it is still difficult to analyze some specific questions, such as modules added or removed, some specific dependencies changes or actual changes in the functionalities.

In their comments, users valued the integration of these visualizations within the software development because they could understand better how the software is organized and how changes affect its organization. However, we have to

improve user interaction, colors, and data representation to emphasize changes. The visualization tool is probably not yet suitable for less experienced team members, but the capacity to automatically display architectural relations in software under development is valuable for software architects and quality assurance engineers.

C. ARCHITECTURE LIMITATIONS

Due to the asynchronous nature of the command design pattern architecture presented in Fig. 3, the software cannot execute commands in accurate timing. In the flight software, commands pass through the invoker and arrive in the receiver where the command is executed. If a command takes considerable time to run, the execution of all other commands in the queue will be delayed. However, this behavior was chosen by design, because microcontrollers usually have very limited resources. Queuing commands for an organized execution helps to control the program flow and reduce errors derived from concurrency, memory usage or CPU load. It is worth noting that a natural extension of this solution is to add multiple receivers to implement a thread pool pattern.

VII. CONCLUSIONS

CubeSat nanosatellites emerged as an accessible methodology to reach space. The accessibility is achieved by reducing the mission cost through standardization, size and weight reduction, and adding agility to the development. It has allowed that actors with limited experience in the topic or far from large space agencies, such as students at universities, research centers, and startup companies, can reach space in short amount of time. However, this flexibility has also added risk. From reports of NASA to engineering thesis, we encountered that flight software design and development is an active concern in the CubeSat community. Software complexity may compromise the space mission success; therefore it is relevant to design and develop high-quality flight software. In CubeSat flight software development, this flexibility may imply that students can come and go to the mission in quarters or semesters, similarly to young engineers in a startup company. This flexibility requires a closer follow up of the code under development. Therefore, our approach moves the efforts to design a software solution that meets important quality criteria and to maintain the proposed software architecture during the development cycle in an agile manner. We propose an *architecture-tracking* tool to maintain control of the flight software architecture on real time during the project development, and in this manner, also monitor the quality of the flight software. We use the development of the SUCHAI CubeSats to show the use of the methodology. We present the requirements for the SUCHAI program flight software and the designed architecture that satisfy them. The architecture is the structure closely followed for the architecture-tracking tool, since it is assumed that if the architecture is followed at all time the quality of the flight software, which is related to the accomplishment of the requirements, by design will be met.

Motivated by a review of a series of CubeSat missions and flight software solutions we selected a set of non-functional and functional requirements that are relevant to these space missions. We identified five quality attributes that a flight software design should consider: i) it should be easily extensible (**Q1**), ii) the flight software should be modular (**Q2**), iii) the flight software should be reliable (**Q3**), iv) the flight software should be portable and reusable (**Q4**), and v) the flight software should scale to the development and operation or large nanosatellite constellation (**Q5**). The functional requirements were extracted from a use case that centered our attention on the necessity of implementing a system capable of executing commands, both remote and self-generated. We therefore implemented a flight software based on the command processor design pattern on top of a three-layer architecture. This three-layer architecture enabled the portability of the software (**Q4**); in fact, we ported the flight software to two operating systems (GNU/Linux and FreeRTOS) and three embedded platforms (AVR32UC3, ESP32, and Nanomind A3200). The command processor pattern generated a solution that can be easily extensible (**Q1**) by adding new commands or new clients. Moreover, clients and commands may be removed without affecting the software's base structure (**Q2**). Unitary test, integration tests, and quality monitoring tools were designed to ensure software reliability (**Q3**).

The proposed solution presents two strengths. First, it is based on three general functional requirements derived from representing the satellite operations as a command executor. We have been successfully able to translate our missions specific functional requirements in the form of commands and commands execution logic. Such a generic and flexible approach may be of interest to the CubeSat developers community. If the expected behavior of a nanosatellite mission matched the command executor model, then this flight software can be adapted for that mission with a level of flexibility inside this model. However, as this behavior is fixed in the upper abstraction layer and by design the command execution is asynchronous, some missions with more demanding requirements may not fit.

Second, as the architecture is inspired by a well-known design pattern, the implemented solution is simple and clear. The main command execution mechanism is implemented once. Then, successive changes in the code are related to the implementation of new commands and to determine when or where these commands are executed. Quality attributes should not be modified. However, to corroborate that the architecture and the quality requirements are effectively accomplished in real code, we developed a set of validation tools. This tools are based on software engineering frameworks and were added in a continuous integration server to track the mentioned software quality criteria along the project history. Our validation methodology includes the following steps:

1) *Visual analysis of the code*: A visual support of the source code helps to detect architecture disruptions in perfectly working code. However, for our purposes, this situation can represent potential errors or a deterioration in software quality. Dependencies, message paths, and differences between commits were visualized to easily determine which components are affected by a source code change.
2) *Automated tests*: Test execution was automated by running unitary and integration test on every commit. Thus we can detect errors early during the development process.
3) *Automated cross-compilation*: Being able to automatically cross-compile code for all the supported platforms is essential to ensuring the software portability at any moment of the development process. With this technique, the collaborators can add new features without having to deal with platform-specific issues. Features tested in GNU/Linux can be integrated later in the embedded platforms by more experienced team members.

We were able to analyze attributes such as extensibility and modularity using a real example of CubeSat under development. That is, extending the features in the actual source code, and determining the effects of these changes over the architectural rules using the proposed visualization framework. Consequently, using the quality assurance methodology, we verified that the flight software source code actually matches the proposed architecture and thus, the quality requirements. The study was useful at showing that the software on board the SUCHAI-1 formerly departed from the desired architecture without being identified as disruptive change at development time. We are also using this approach in the development of the SUCHAI-2 and -3 missions; thus, we can control and track the quality of the software at any point of the development history. At the moment the same flight software solution is being used in four CubeSats and three radiosonde missions at the University of Chile, so we see a scalability potential for using this flight software in nanosatellite constellations.

The flight software and the visualization tools were developed as open source projects. With this decision we not only expect to the community to revise and improve our work, but also we expect to facilitate and make more robust the development of Cubesat-based missions, especially those of new actors. The source code of the SUCHAI flight software can be found in the following GitHub repository https://github.com/spel-uchile/SUCHAI-Flight-Software, the visualization history in the following link http://data.spel.cl/, and the continuous integration system is hosted in the following address https://jenkins.spel.cl/.

VIII. ACKNOWLEDGEMENT

We would like to acknowledge the great work and commitment by entire SPEL team, as well as the support of the Faculty of Physical and Mathematical Sciences at the University of Chile. Special acknowledgments to the contributors of the SUCHAI Flight Software: Tamara Gutiérrez, Ignacio Ibañez, Tomás Opazo, Diego Ortego and Matías Ramirez. We are grateful to Milton Mamani for helping design the visualization. We thank Object Profile and Lam Research for partially sponsoring the work presented in this paper. We gratefully thank Renato Cerro for reviewing an early draft of this article. This work has been partially supported by the grants Fondecyt 1151476, Anillo ACT1405, Fondequip EQM150138 and CONICYT-PCHA/Doctorado Nacional/2016-21161016.

References

[1] S. Lee, A. Hutputanasin, A. Toorian, W. Lan, R. Munakata, J. Carnahan, D. Pignatelli, and A. Mehrparvar, "Cubesat design specification rev. 13," tech. rep., The CubeSat Program, Cal Poly San Luis Obispo, US, 2014.

[2] C. Boshuizen, J. Mason, P. Klupar, and S. Spanhake, "Results from the Planet Labs Flock Constellation," in AIAA/USU Conference on Small Satellites, aug 2014.

[3] D. L. Dvorak, "NASA Study on Flight Software Complexity," in AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference, (Reston, Virigina), p. 264pp, American Institute of Aeronautics and Astronautics, apr 2009.

[4] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in Proceedings of the International Conference on Dependable Systems and Networks, pp. 1–8, IEEE, jun 2013.

[5] P. Fiala and A. Vobornik, "Embedded microcontroller system for PilsenCUBE picosatellite," in 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pp. 131–134, IEEE, apr 2013.

[6] A. van den Berg, "Fault-tolerant on-board computer software for the del fi-n3xt nanosatellite," Master's thesis, Delft University of Technology, Delft, Netherlands, August 2012.

[7] S. Johl, E. Glenn Lightsey, S. M. Horton, and G. R. Anandayuvaraj, "A reusable command and data handling system for university cubesat missions," in IEEE Aerospace Conference Proceedings, pp. 1–13, IEEE, mar 2014.

[8] M. Schmidt and K. Schilling, "An extensible on-board data handling software platform for pico satellites," Acta Astronautica, vol. 63, pp. 1299–1304, dec 2008.

[9] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the KySat CubeSat bus," in IEEE Aerospace Conference Proceedings, pp. 1–15, IEEE, mar 2009.

[10] C. Mitchell, J. Rexroat, S. A. Rawashdeh, and J. Lumpp, "Development of a modular command and data handling architecture for the KySat-2 CubeSat," in IEEE Aerospace Conference Proceedings, pp. 1–11, IEEE, mar 2014.

[11] G. Manyak and J. M. Bellardo, "Polysat's next generation avionics design," in 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, pp. 69–76, Aug 2011.

[12] I. Sünter and SOFTWARE, Software for the Estcube-1 Command and Data Handling System. PhD thesis, University of Tartu, 2014.

[13] D. Schor, J. Scowcroft, C. Nichols, and W. Kinsner, "A command and data handling unit for pico-satellite missions," in Canadian Conference on Electrical and Computer Engineering, pp. 874–879, 2009.

[14] S. A. Asundi and N. G. Fitz-Coy, "Design of command, data and telemetry handling system for a distributed computing architecture CubeSat," in IEEE Aerospace Conference Proceedings, pp. 1–14, IEEE, mar 2013.

[15] C. Araguz López, Towards a modular Nano-Satellite Software Platform: Prolog Constraint-based Scheduling and System Architecture. PhD thesis, Universitat Politècnica de Catalunya, sep 2014.

[16] M. A. Normann and R. Birkeland, Software Design of an Onboard Computer for a Nanosatellite. PhD thesis, Norwegian University of Science and Technology, 2016.

[17] M. Pagnamenta, Rigorous software design for nano and micro satellites using BIP framework. PhD thesis, École polytechnique fédérale de Lausanne, 2014.

[18] S. Nakajima, R. Funase, S. Nakasuka, S. Ikari, M. Tomooka, and Y. Aoyanagi, "Command Centric Architecture (C2A): Satellite Software Architecture with a Flexible Reconfiguration Capability," in 68th International Astronautical Congress (IAC), (Adelaide, Australia), 2017.

[19] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in AIAA/USU Conference on Small Satellites, aug 2016.

[20] R. Plauche, "Building modern cross-platform flight software for small satellites," in AIAA/USU Conference on Small Satellites, Aug 2017.

[21] A. Mavridou, E. Stachtiari, S. Bliudze, A. Ivanov, P. Katsaros, and J. Sifakis, "Architecture-Based Design: A Satellite On-Board Software Case Study," in Formal Aspects of Component Software. FACS 2016., pp. 260–279, Springer International Publishing, 2017.

[22] C. Araguz, M. Marí, E. Bou-Balust, E. Alarcon, and D. Selva, "Design Guidelines for General-Purpose Payload-Oriented Nanosatellite Software Architectures," Journal of Aerospace Information Systems, vol. 15, pp. 107–119, mar 2018.

[23] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew, "Verifying architectural design rules of the flight software product line," in Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings, pp. 161–170, Carnegie Mellon University, 2009.

[24] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, and B. Medina, "Architecture-based unit testing of the flight software product line," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6287 LNCS, pp. 256–270, Springer, Berlin, Heidelberg, 2010.

[25] C. Gonzalez, C. Rojas, A. Becerra, J. Rojas, T. Opazo, and M. Diaz, "Lessons learned from building the first Chilean Nano-satellite : the SUCHAI project," in AIAA/USU Conference on Small Satellites, aug 2018.

[26] M. A. Diaz, J. C. Zagal, C. Falcon, M. Stepanova, J. A. Valdivia, M. Martinez-Ledesma, J. Diaz-Peña, F. R. Jaramillo, N. Romanova, E. Pacheco, M. Milla, M. Orchard, J. Silva, and F. P. Mena, "New opportunities offered by Cubesats for space research in Latin America: The SUCHAI project case," Advances in Space Research, vol. 58, pp. 2134–2147, nov 2016.

[27] M. Glinz, "On non-functional requirements," in 15th IEEE International Requirements Engineering Conference (RE 2007), pp. 21–26, Oct 2007.

[28] M. Cho, M. Hirokazu, and F. Graziani, "Introduction to lean satellite and ISO standard for lean satellite," in 2015 7th International Conference on Recent Advances in Space Technologies (RAST), pp. 789–792, IEEE, jun 2015.

[29] IEEE, "IEEE Standard for a Software Quality Metrics Methodology. IEEE Std 1061-1992," tech. rep., 1993.

[30] J. Bouwmeester and J. Guo, "Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology," Acta Astronautica, vol. 67, pp. 854–862, oct 2010.

[31] J. Javanainen, Reliability evaluation of Aalto-1 nanosatellite software architecture. PhD thesis, Aalto University, feb 2016.

[32] J. Guo, J. Bouwmeester, and E. Gill, "In-orbit results of Delfi-n3Xt: Lessons learned and move forward," Acta Astronautica, vol. 121, pp. 39–50, apr 2016.

[33] S. Nakasuka, N. Sako, H. Sahara, Y. Nakamura, T. Eishima, and M. Komatsu, "Evolution from education to practical use in University of Tokyo's nano-satellite activities," Acta Astronautica, vol. 66, pp. 1099–1105, apr 2010.

[34] G. D. Manyak, Fault Tolerant and Flexible CubeSat Software Architecture. PhD thesis, California Polytechnic State University, San Luis Obispo, San Luis Obispo, California, jun 2011.

[35] R. Funase, E. Takei, Y. Nakamura, M. Nagai, A. Enokuchi, C. Yuliang, K. Nakada, Y. Nojiri, F. Sasaki, T. Funane, T. Eishima, and S. Nakasuka, "Technology demonstration on University of Tokyo's pico-satellite "XI-V" and its effective operation result using ground station network," Acta Astronautica, vol. 61, pp. 707–711, oct 2007.

[36] M. Pessans-Goyheneix, J. Bønding, M. Burchard, T. Kasper, and F. Jensen, "Software Framework for Reconfigurable Distributed System on Aausat3," tech. rep., Aalborg University, 2008.

[37] S. Corpino and F. Stesina, "Verification of a CubeSat via hardware-in-the-loop simulation," IEEE Transactions on Aerospace and Electronic Systems, vol. 50, no. 4, pp. 2807–2818, 2014.

[38] S. Busch, P. Bangert, S. Dombrovski, and K. Schilling, "UWE-3, in-orbit performance and lessons learned of a modular and flexible satellite bus for future pico-satellite formations," Acta Astronautica, vol. 117, pp. 73–89, 2015.

[39] MIRA Ltd, "MISRA-C:2004 Guidelines for the use of the C language in Critical Systems," tech. rep., Motor Industry Software Reliability Association, 2004.

[40] G. J. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code," Computer, vol. 39, pp. 95–99, jun 2006.

[41] National Academies of Sciences, Engineering, and Medicine, Achieving science with CubeSats: Thinking inside the box. National Academies Press, 2016.

[42] R. J. Barnett, "Oneweb non-geostationary satellite system: Technical information to supplement schedule s - attachment to fcc application sat-loi-20160428-00041," tech. rep., 2016.

[43] M. Albulet, "Spacex non-geostationary satellite system: Technical information to supplement schedule s - attachment to fcc application sat-loa-20161115-00118," tech. rep., 2016.

[44] I. Sommerville, Software Engineering. Delhi, 2006.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[46] F. Buschmann, R. Meunier, H. Rohnert, P. S. Stal, and A. Michael, Pattern-Oriented Software Architecture: a system of patterns, vol. 1. Wiley, 1996.

[47] M. Petre, "Why looking isn't always seeing: readership skills and graphical programming," Communications of the ACM, vol. 38, pp. 33–44, jun 1995.

[48] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, Deep Into Pharo. Square Bracket Associates, 2013.

[49] A. Bergel, Agile Visualization. LULU Press, 2016.

[50] M. Lanza and S. Ducasse, "Polymetric views - A lightweight visual approach to reverse engineering," IEEE Transactions on Software Engineering, vol. 29, pp. 782–795, sep 2003.

CARLOS E. GONZALEZ obtained his B.S. degree in electrical engineering from the University of Chile, Santiago, Chile, in 2014. He is currently pursuing the Ph.D. degree in electrical engineering at University of Chile, Santiago, Chile. From 2011 to 2014 he worked at the Space and Planetary Exploration Laboratory (SPEL) of the University of Chile developing the first nanosatellite of the country. His work focused in the development of the flight software and communication system of the SUCHAI satellite. From 2014 to 2016 he worked as a researcher engineer at the Advanced Mining Technology Center (ATMC) of the University of Chile developing software for geostatistical applications in the mining industry. Since 2015 he is the lecturer of the Computer Networks course at the University of Santiago of Chile (USACH).

CAMILO J. ROJAS obtained his B.S degree in Electrical Engineering from the University of Chile in 2012. Currently achieving his Master degree in Computer Science at University of Chile. From 2011 to 2012 he worked in the first Chilean nanosatellite project, SUCHAI project, being a developer in the communication group. From 2012 to 2016 worked at Synopsys Chile as a software developer in TCAD group for the main visualization tool used in the semiconductor market. Currently his is working as a grad student in the Advanced Laboratory for Geostatistical Supercomputing (ALGES) of the University of Chile and as a flight software developer in the Space and Planetary Exploration Laboratory (SPEL) of the University of Chile.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2927931, IEEE Access

ALEXANDRE BERGEL obtained his PhD in Computer Science from the University of Bern, in 2005. Since 2009, he is Associate Professor and researcher at the University of Chile. He and his collaborators carry out research in software engineering. His effort is about designing tools and methodologies to improve the overall performance and internal quality of software systems, by employing profiling, visualization, and artificial intelligence techniques. Alexandre has also a strong interest in applying his research results to industry. Several of his research prototypes have been turned into products and adopted by major companies in the semi-conductor industry and certification of critical software systems. Alexandre authored the book Agile Visualization and co-authored the book Deep Into Pharo.

MARCOS A. DIAZ received his Electrical Engineering degree in 2001 from University of Chile, his M.S. and Ph.D. degrees in Electrical Engineering in 2004 and 2009, respectively from Boston University, USA. He is an Assistant Professor in the Electrical Engineering Department at University of Chile, Santiago, Chile. His research interests are related to the study of ionospheric turbulent plasma, incoherent scatter radar techniques, low-frequency-radio-astronomy/space instrumentation and nano-satellite technologies. He is the responsible of the Space and Planetary Exploration Laboratory, a multidisciplinary Laboratory located in the Faculty of Physical and Mathematical Sciences at University of Chile, where the nanosatellite-based space program at the University is being developed.