© 2018 American Institute of Aeronautics and Astronautics.

Carles Araguz, Marc Marí, Elisenda Bou-Balust, Eduard Alarcon, and Daniel Selva. "Design Guidelines for General-Purpose Payload-Oriented Nanosatellite Software Architectures", Journal of Aerospace Information Systems, Vol. 15, No. 3 (2018), pp. 107-119.

https://doi.org/10.2514/1.I010537

Design Guidelines for General-Purpose Payload-Oriented Nano-satellite Software Architectures

Carles Araguz*,Marc Marí,Elisenda Bou-Balust,Eduard Alarcon

Nano-Satellite and Payload Laboratory, Dept. of Electronics Engineering, Technical University of Catalonia – UPC BarcelonaTech, 08034 Barcelona, Spain

Daniel Selva

Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY

Abstract

Despite their limited lifespan and reduced cost, nano-satellite missions have proved to be suitable platforms for Earth observation, scientific experiments and technology demonstration. During the last years, the number of nano-satellite missions has noticeably increased, posing the need to improve several system characteristics to ultimately endorse the full potential of this class of spacecraft. In this context, this paper presents three design guidelines that can be applied in nano-satellite software in order to improve the system robustness, modularity and autonomy. The design guidelines presented in this paper, namely, hierarchy-enabled robustness, payload-oriented modularity, and on-board planning capabilities, are complemented with a structured review of complementary software techniques and architectural concepts that have been found in the literature. The paper justifies that these system-wide qualities are some of the most critical when designing flight software for CubeSat-like spacecraft and explores how they can improve mission performances and operability, enhance the system's tolerance to failures and ease the development cycles. Finally, this paper illustrates the application of the design guidelines by detailing the on-board software architecture for the ^{3}Cat-1, a CubeSat program carried out at the Nano-Satellite and Payload Laboratory of the Technical University of Catalonia (UPC BarcelonaTech).

1. Introduction

Nano-satellites have become an affordable alternative for many companies, research organizations and universities to access the space market, both as consumers and providers. Usually deployed in Low-Earth Orbits, nano-satellites have proven to be suitable platforms for technology demonstration [1], a variety of Earth observation and remote sensing purposes, science and research (e.g. [2]) and many other space applications such as low-power communications or maritime activity surveillance (e.g. [3]).

*Corresponding author:
E-mail address: carles.araguz@upc.edu (Carles Araguz)

Either adopting the CubeSat design philosophy and standardized structure, as in the 3U-based FLOCK constellation by Planet Labs [4], or designing spacecraft busses that take up less than 60 cm per side[1] like NASA's CYGNSS constellation [5], nano-satellite platforms have already been adopted by agencies, small and large corporations and have have been developed under many educational programs since the appearance of the CubeSat standard. While the latter types of missions tend to be fully designed, implemented and operated by heterogeneous teams at universities and are generally less demanding in terms of accuracy and reliability, the vast presence of university-developed nano-satellite missions is a clear sign of the ongoing democratization of space and the constant exploration of small spacecraft's science return capabilities. On the other hand the interest and adoption of these types of platforms by the industry (e.g. [6, 7]) evidences a clear paradigm shift and suggests a complexity increase for future mission architectures based on nano-satellite technologies.

Albeit this situation has led to the development of multiple successful, monolithic nano-satellite missions, lately, the adoption of this class of spacecraft has also been considered especially favorable for the development of new mission architectures such as fractionated spacecraft, satellite constellations and swarms [8]. The combination of several instruments hosted at different nano-satellites has been envisaged as an enabler for new Earth observation missions with enhanced performance and improved system qualities [9].

In this scenario, some companies have already been offering software components, hardware modules and complete subsystems that are compliant with the de facto standard (i.e. CubeSat Units), ranging from complex Attitude Determination and Control Subsystems (ADCS), to Electrical Power Supplies (EPS), robust communication protocols, low-power on-board computers or Real-Time Operating Systems (RTOS). These and many other CubeSat-compliant commercial components, facilitate the development and integration of spacecraft and remove the burden of designing and testing some critical subsystems and modules. Thus, apart from coping with the complex task of integration, most nano-satellite developers ultimately focus on the development of mission-specific payloads and, most importantly, the design and implementation of custom flight software that controls the spacecraft at device- and system-level.

Given that software is usually understood as the final architectural element to achieve the desired functionality, less attention has been placed on software-related issues during the emergence and consolidation of the CubeSat era. However, software and its architectural characteristics can be critical for the management of the mission; their correctness severely affects the functionality of the spacecraft. As a matter of fact, designing proper software architectures is also essential to achieve system-wide qualities such as reliability and performance, and should not be understood as the mere fact of writing functionally correct programs.

In this context, this paper poses the need for improvement and explores the qualities and characteristics

[1]NASA's CYGNSS spacecraft measures 18 x 42 x 60 cm when stowed, although this volume mostly accounts for the GNSS-R antenna and solar panels.

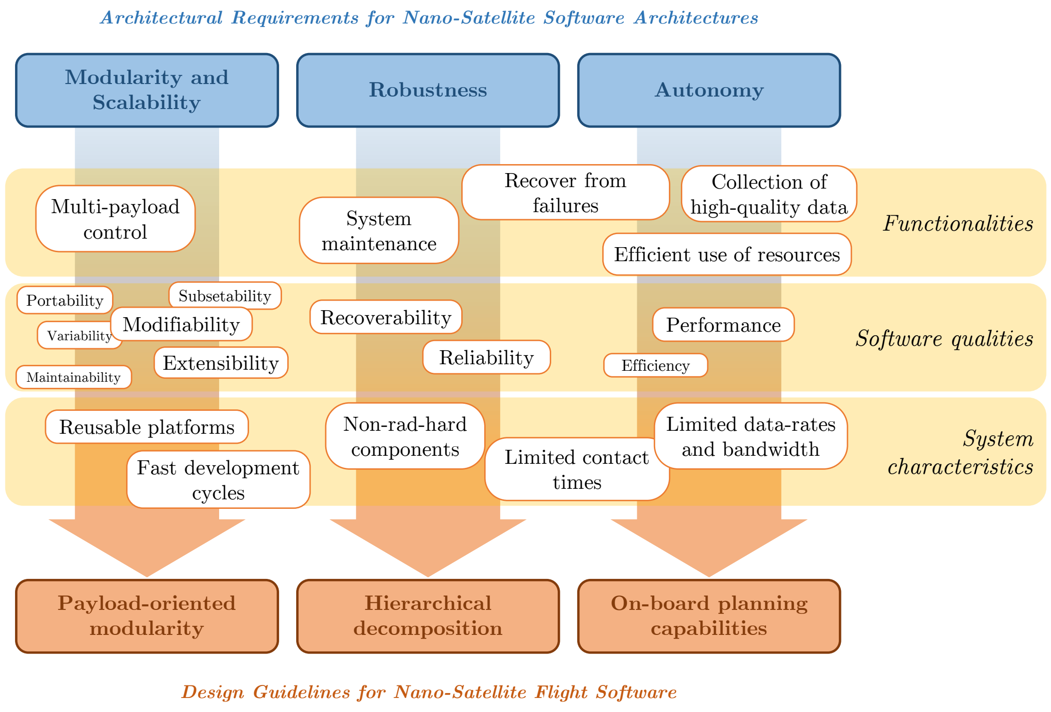

Figure 1: Visual summary of common desired functionalities, qualities and characteristics of nano-satellites, critical architectural requirements for their on-board software, and how they map to three design guidelines.

of nano-satellite systems. By identifying critical functionality and architectural requirements, this paper motivates the application of a design methodology for new designs that is composed of three essential guidelines, namely:

1. *Payload-oriented modularity*: emphasizes the importance of proper encapsulation and generalization of low-level components in order to adapt software architectures to the needs of multi-payload missions and fast development cycles.

2. *Robustness through hierarchical decomposition*: aims at structurally reducing error propagation and intends to minimize the complexity of critical system control parts by decoupling them from hardware and low-level modules.

3. *On-board planning capabilities*: provides a set of minimum components that can enhance the autonomy of a spacecraft by virtue of automatic generation of mission plans and robust execution of tasks.

This set of design guidelines, summarized in Figure 1, are presented as generic overarching characteristics rather than implementation features. Thus, they can be applied vertically throughout a whole flight software framework. Furthermore, due to the fact that specific industry standards are not considered in this study, these design guidelines can be embraced both by educationally-based programs and by industries developing their spacecraft, contributing, thus, to the foundations of future-generation nano-satellite software architectures.

This paper is organized as follows: Section 2 and 3 identify the critical qualities and features that modern nano-satellite software should improve. Section 3 explores techniques to do so by revisiting and structuring knowledge hitherto presented in literature. Section 4 derives the set of generic design guidelines which can be adopted at architectural levels and which improve the selected system qualities. Finally, Section 5 illustrates the application of these design criteria with an instance of those guidelines in the flight software architecture of the ^{3}Cat-1 nano-satellite mission.

2. Identifying the architectural requirements for next-generation software architectures

Determining the essential requirements for nano-satellite flight software requires the assessment of both their functional traits, the particularities of this type of systems, and the desired all-encompassing qualities. The impact of these three aspects upon the design process is probably common to that of many engineering fields, and their interaction needs be taken into consideration just as much as they need be individually considered. Functional requirements essentially describe the behavior of the system (internal and external) and can easily draw specific structural design requirements: functions can easily be decomposed into blocks and their interrelationships. On the other hand, the specific system characteristics, or its context, will also reveal indirect design requirements that should be well studied at design time. Not only these relate to the limiting conditions under which the system operates (e.g. intermittent or continuous operation, influenced by external factors that may cause failures, setting a maximum number of concurrent operations) but they also encompass particular details of the development process, such as: whether they need to be produced in mass, or not; the types of devices on top of which the software will run; how much of the design will need to be changed and which parts will suffer greater modifications; etc. The number of possible functions and specific system characteristics can be vast and is out of the scope of this study. Instead, the present analysis is interested in common aspects that do apply in most nano-satellite missions and which can translate to generic design requirements.

Indeed, this set of generic requirements is very much motivated by functional characteristics, system limitations and external conditions. However, these factors also enforce high-level attributes of a software architecture, which can also be grouped and studied under the term of *quality attributes*. Accordingly, a software architecture should not only define the system in terms of tangible actions, relationships or functionalities but it should also play a significant role in achieving these system-wide quality attributes. Designing a suitable architecture will allow or preclude just about all of a system's characteristics, thereby leading the goodness of a software architecture to strongly affect the integrity of the whole system.

The IEEE Standard for a Software Quality Metrics Methodology [10] defines software quality as the degree to which the software possesses a desired combination of attributes. These non-functional attributes of a software system (e.g. reliability, performance, usability) can map to specific, yet high-level, requirements and may often be intertwined with each other.

The set of quality attributes one can use to assess a particular software architecture differs depending upon the source. So much so that lexically similar or identical qualities can have different names and the same quality can be found with slightly different definitions. The considered attributes and their definitions in this paper are taken from the standards in [11–14] and references [15, 16]. Note, however, that because they relate to qualitative aspects, assessing them quantitatively is often a very subjective exercise that has not tackled neither in the cited works nor in this analysis. In order to minimize semantic ambiguities and provide a more generic set of requirements, this paper proposes three groups of quality attributes that encompass many of the specific ones discussed in the references. The requirements proposed in this paper, justified in detail in the following sections, are: (a) robustness, (b) modularity and scalability and (c) autonomy. These quality attributes will ultimately be mapped into three independent design rules later in this paper.

Aside from possible inter-dependencies, software quality attributes may be conceptually bound to the desired functionality and external limitations of a system. In other words, some system limitations and functional requirements will force some of this qualities to become actual requirements. As an illustration of the latter statement, consider a case in which a software needs to process large volumes of data (function) but is forced to run in a computationally-limited hardware (system characteristic). While, in this case, the software architecture would require a certain *performance*, if the software would run on several different platforms, one would say that the software needs to have high *portability*. This conceptual binding between functionalities, system characteristics and the high-level qualities of a software, are graphically represented in Figure 1, for the groups of attributes proposed above.

2.1. Robustness

Space applications are subject to countless sources of failures. While the effects of ionized particles in the on-board semiconductors, such as Single Event Upsets (SEU) and Single Event Latchups (SEL) are one of the most common sources of errors, other situations like one-time subsystem malfunctions, power or communication failures can cause a variety of run-time errors. In order to protect their microcontrollers and memories against SEU's and SEL's, large spacecraft are often equipped with radiation-hardened devices that can withstand greater doses of ionizing radiation. Contrarily, nano-satellite designs hardly include such devices, usually owing to the spacecraft limited power and mass budgets and sometimes due to the habitual use of regular COTS components.

Similarly, large satellites also combine the use of rad-hard technology with sophisticated real-time operating systems, hypervisors and middleware which present reliability guarantees and provide the foundations for robust software environments. The latter types of products, however, are not restricted in nano-satellite developments. A good example of these can be found in NASA/GSFC's Core Flight Executive and Core Flight Software[2] (cFE and cFS), an open-source middleware which is available for some open-source kernels[3] (e.g. RTEMS, Linux) and which can be used to develop flight applications. The cFS/cFE middleware is a comprehensive flight software framework that extends the Operating System and provides common services and a myriad of re-usable modules. Adopting these may ameliorate the robustness of the system since this products have been exhaustively tested and verified [17]. Regardless of this suite already being tested in NASA's nano-satellite missions [18], its adoption is still not broad enough and many current developments are still implemented on top of simpler, commonly known operating systems (e.g. FreeRTOS or standard Linux kernels), which lack most of the reliability guarantees of additional, space-qualified middleware.

In these situations, the mission software should be able to withstand the system's failures and correct them. From an architectural standpoint, not considering qualities like *recoverability* or *reliability* during the software design process poses a risk to the mission and could become one of the causes of a global breakdown. While the reliability of an architecture is related to the ability to perform the required functions without failures or within a bonded failure rate, recoverability emphasizes how good the recovery strategies are. In order to recover from a fault or unexpected state, one may argue that the architecture needs to be designed with robust system state control and some kind of error detection mechanism. Because of that, this evaluation framework considers critical to implement the available architectural methods and alternatives to achieve robust software, also in nano-satellite programs.

2.2. Software modularity and scalability

During the last years, developing, launching and operating high-density constellations of nano-satellites started to become a reality. Ventures like the one started by Planet Labs have planned to operate constellations of up to 200 homogeneous units [4] in order to offer Earth imagery at medium-resolutions (3-5 m) with daily revisit times. The capabilities of satellite constellations consisting of many nano-satellite units are promising and suggest the need for modular architectures, also from a software viewpoint. As a matter of fact, next generation constellations should not necessarily involve identical units orbiting at different orbits but could also encompass a set of heterogeneous nano-satellites orbiting closer and communicating with one another. In that scenario, software architectures not only need be replicable but shall be variable enough to attain the control of a diversity of payloads.

Simultaneously, many nano-satellite programs tend to continue their activities after a successful mission and develop new generations of spacecraft based on their previous designs. In this context, designing reusable

[2]https://cfs.gsfc.nasa.gov

[3]Please note that at the time of writing this paper, FreeRTOS support for cFS/cFE was already in development but had not been officially released as part of NASA's Operating System Abstraction Layer (OSAL).

systems does have a significant importance in order to reduce, even more, the development times of future nano-satellite units.

As technology advances, new, smaller, less power-consuming and more capable devices and modules will surface. Nano-satellites will, then, increase their payload capacities and will likely require flight software capable of controlling and interfacing more subsystems. The need for scalable and flexible software architectures were modules can be added, changed or removed without affecting the core of the architecture therefore becomes evident. In this respect, several attributes can be studied to asses how complex it is to modify an architecture to some extent. Terms such as *variability*, *extensibility* and *subsetability* reflect the ease with which a software architecture can be modified to produce new designs that differ in specific, preplanned ways and assess the required actions to do so. Similarly, *maintainability* or *modifiability* also express the ease with which a software system or component can be modified to correct faults, improve some characteristic or adapt to a changed environment. When the changes are exogenous to the software architecture itself, one can study the *portability* of the design by assessing how complex it is to migrate it from one platform[4] to a different one. With more or less emphasis in each of them, this set of qualities are deemed essential in this study and will be considered when deriving specific design guidelines in the sections that follow.

2.3. Spacecraft autonomy

There are many different factors which suggest that nano-satellites be provided with a certain degree of autonomy. To begin with, observability in satellites can be extremely constrained. Depending on the orbit altitude and inclination and the location of the ground stations, satellites in Low Earth Orbits (i.e. most nano-satellites) can establish communication links in the order of up to 4-5 times per day, with durations in the range of 5 to 10 minutes, approximately. Assuming good elevation conditions for the ground station antennae, nano-satellite operators might be able to communicate with their spacecraft during 30-45 minutes every 24 hours if occasional link deterioration caused by environmental factors is not considered. Consequently, the amount of information that can be downloaded from a LEO satellite is extremely restricted, let alone the limited data rates often found in nano-satellite systems.

These communication restrictions may preclude mission operators from reacting to unexpected failures with agility and could lead the satellite to remain in safe modes for long periods after an error is detected. Intermittent communications thus worsen the spacecraft performance if science opportunities are missed during the latter situation. Conversely, autonomous spacecraft that can replan their activities not only will improve the spacecraft's data acquisition capabilities but will also ameliorate the mission's robustness.

On the other hand, limited telemetry bandwidth could prevent the retrieval of fine-grained states of the satellite. When satellite actions and resource allocations are planned in the ground segment, not knowing the

[4]Here the term "platform" can refer to either the underlying hardware modules, or the CPU architecture, or the OS.

state of the spacecraft with enough accuracy can negatively affect the way they are computed. Combining spacecraft state uncertainties with the uncertainties inherently found in space environments may, in addition, turn the plan generation into a very complex endeavor.

Furthermore, autonomous spacecraft can also deliver higher quality data if they are provided with algorithms that intelligently optimize the data collection and download. This capability is not new for large spacecraft and has been demonstrated in the past for several satellite missions. An example of this is NASA's Earth Observing One (EO-1), a spacecraft which performed on-board data analysis and replanning to optimize the volume of data downloaded to ground [19]. The three-tiered software architecture in EO-1 encompassed a scheduler module (CASPER) that was able to replan activities, including downlink, based on science observations in the previous orbit cycles. Although computing resource allocation and performing data analysis on-board demands higher computational capabilities, the same on-board planning software was successfully integrated in the IPEX nano-satellite mission in 2013 [20], demonstrating that this class of satellites can also support and benefit from the sophisticated algorithms present in autonomy systems.

Notwithstanding the fact that the commented autonomous capabilities are common in large satellite missions, many nano-satellite operation approaches are still relying upon the interaction of spacecraft with ground segment controllers. When nano-satellites pass over their ground stations, they receive sequences of time-tagged telemetry commands whose execution is statically scheduled at ground. Modern approaches are those implementing *goal-based* operations, where ground operators only modify the mission *goals* and allow the spacecraft to autonomously schedule its activities to meet the current goals [21, 22]. Mission goals inherently encapsulate complex and flexible command sequences that will be decomposed on-board, and are not accompanied by a fixed execution time. This type of approaches, in which nano-satellites would be more autonomous, improve the mission *performance* by allowing the spacecraft to optimize the timeline of actions not only based on the state of resources and subsystem but also with a given degree of awareness of captured data quality. Nano-satellites with instruments that can only operate under certain conditions (e.g. optical imagers are generally constrained by lighting and cloud-coverage conditions) could autonomously decide when to enable their instruments based on predictions (e.g. lighting conditions) and on-board analysis of data (e.g. cloud coverage), thus saving power and storage efficiently.

Having explored the system characteristics, qualities and common functional aspects, this paper proposes this framework of three essential requirements (robustness, modularity and scalability, and autonomy) to be applied during the design process of future nano-satellite software. The items presented in the framework have essentially arisen from the technological context and current trends in the small spacecraft community: while CubeSat-based systems have proven to be a time- and cost-effective alternative to develop high-performance, complex space systems, the number of nano-satellite programs is growing incessantly. All the aforementioned architectural requirements are achievable through software engineering efforts and encompass many of the quality attributes found in architecture evaluation methods.

Whereas there are many systems described in the literature which improve most of the critical qualities (reliability, flexibility, autonomy, performance, etc.) their descriptions and designs have been presented and addressed from their particular mission standpoints, preventing their architectures, components and techniques to be generally applied in different contexts. It is the purpose of this paper to explore some of these techniques and designs and to derive a set of design guidelines that satisfy the requirements presented in this section and which can be generally applied in new software architectures for nano-satellites.

3. Review of Current Solutions

This section gathers a compendium of architectural concepts and design techniques that are present or have been applied in successful programs. While some of the items of this list are basic concepts with which most software engineers will be familiar, all of them can be concurrently embraced at the design phase as a means to improve some of the essential quality attributes for nano-satellite flight software.

3.1. Process isolation and protected shared resources

Time and Space Partitioning (TSP) are well-known techniques in the aviation and space industries to deliver robust software. TSP kernels and middleware allow one computer to be used for many different applications and the controlled use of the system's shared resources (i.e. memory, I/O devices...) Applications executed using a TSP approach have their memory regions protected from the rest of the processes and are executed within deterministic time slots managed by a global hypervisor running on top of the OS. While Inter-Partition Communication is reliably managed and guaranteed by intermediate layers, this approach allows high-level applications to be executed seamlessly without affecting other computer components upon their failure.

Applying time and space protection for the applications running in spacecraft is a common and recommended approach that can be found in many spacecraft designs [23]. Nonetheless, it may require the utilization of specific RTOSes or middleware. The idea of process isolation, however, is also implemented in widely known kernels like Linux. These operating systems are, on the contrary, much more available and known than specific TSP products and they naturally provide process isolation (i.e. a virtual address space for each process). The use of Linux in small satellite developments is not new (e.g. [20, 24–28]) and also owes to the inherent use of readily available COTS components (i.e. on-board computers). In Linux-based designs, however, the mechanisms to control the access to shared resources (e.g. storage devices, communication ports, etc.) should be carried out separately, since most Linux drivers are not designed to provide such feature.

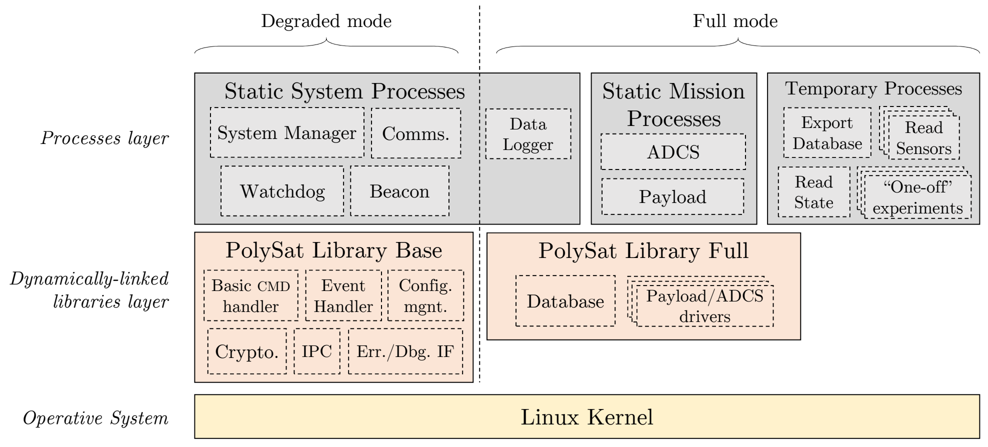

Figure 2: PolySat's Second Generation Bus software architecture (extracted from [29] and adapted)

3.2. Real-Time Operating Systems

The utilization of real-time software in the nano-satellite community is highly accepted and recommended. Programming flight software applications in real-time environments is essential to guarantee the execution of critical processes. While priority-based real-time systems implement reliable scheduling algorithms that prevent task priority inversions, inter-task communication and synchronization services provided by real-time kernels also allow the deployment of complex architectures.

Most modern RTOSes support many of the processor architectures found in spacecraft computers (e.g. PowerPC, ARM, SPARC...) The availability of hard-real-time OSes ranges from industry renowned products like RTEMS[5], VxWorks[6], QNX[7], LynxOS[8] and so on, to small-footprint, free and/or open-source alternatives such as FreeRTOS[9] or µC/OS-III[10]. On the other hand, in the list of real-time alternatives one can also include Linux patches, such as PREEMPT_RT[11] or Xenomai[12]. These options, which provide soft-real-time capabilities to standard Linux kernels (e.g. priority-based preemptive scheduling), can also be suitable for nano-satellite developments although their timing capabilities (i.e. latencies) are not comparable to specialized hard-real-time products.

3.3. De-embeddable core and safe devices

Architectural approaches to improve software robustness can be found in multiple nano-satellite designs. An easily applicable example can be found in the flight software developed at California Polytechnic State

[5]https://www.rtems.org
[6]https://www.windriver.com/products/vxworks
[7]http://www.qnx.com
[8]http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos
[9]http://www.freertos.org
[10]https://www.micrium.com/rtos
[11]https://rt.wiki.kernel.org
[12]https://xenomai.org

University, for their series of CPx nano-satellites. Figure 2 shows the two-tiered PolySat's software architecture, consisting of the *Processes* and *Libraries* layers, deployed on top of a Linux kernel. The processes layer encompasses the so-called *static* processes (i.e. processes which are always active) and the *temporary* processes (i.e. are launched on-demand). Depending on their functionality and criticality, static processes are sub-categorized into *system* or *mission* processes, clearly identifying the main platform modules. In addition, the libraries layer, which provide services and hardware interfaces to the processes layer, is also logically divided into a set of basic libraries ("PolySat Library Base") plus an extension set ("PolySat Library Full").

Their reusable software architecture has been the main controlling software on-board CPx spacecraft and is characterized by presenting two modes of operation, namely, *degraded-* and *full-mode*. In degraded-mode, the on-board computer only runs critical processes which do not access devices that are sensitive to radiation damage (i.e. NAND storage device). These critical processes implement the minimal functionality for the satellite to be operable and are responsible to switch to *full-mode* once the contents of the NAND device have passed an integrity test.

Designing software architectures with de-embeddable cores that require a minimum set of hardware components to run, may allow ground operators to keep control of the spacecraft even when parts of the system are unusable. In the lack of redundant systems, this kind of techniques can enhance the overall robustness of the system and ameliorate the lifespan of the spacecraft.

3.4. FDIR methodology

Fault Detection, Isolation and Recovery techniques are quite spread among space applications and have recently landed in the nano-satellite community. FDIR systems externally monitor system variables and infer the occurrence of errors by checking their expected values, usually against a predefined model. In the event of failures, FDIR systems detect their severity and apply some actions to isolate, circumvent and solve the errors, whenever possible. Implementing safe modes where the spacecraft can safely remain upon the occurrence of errors is a common technique that allows ground operators to return the system to a desired state and may prevent loss of contact.

Despite the high computational load required to run complex FDIR systems, there have been nano-satellite missions that considered them to some extent. On one hand, Technical University of Delft has applied FDIR analysis on their DelFFi program [30] by studying how to detect and isolate failures in several subsystems and devices (e.g. deployables, UART, I^2C, memories). Their analysis resulted in a set of procedures and rules to trigger state transitions to safe modes and/or to signal the errors.

Among the active small satellite initiatives, ESA's 3U CubeSat OPS-SAT encompasses a dedicated FDIR computer which monitors each payload board through a modular controller [25]. Apart from the ability to monitor houskeeping data coming from the OBC and reacting to a small set of telemetry commands, the

FDIR computer is able to circumvent Single Event effects (i.e. SEU, SEL) by electrically isolating the payload boards from the system bus.

3.5. Dynamically-linked libraries

Segmented software implementations which consist of a set of programs, libraries and drivers can be partially updated by only replacing some of their fractions. While shared libraries offer the possibility to encapsulate reusable code outside the kernel that can be loaded or called by several programs, the fact that they are easily replaceable significantly increases the update-ability and modularity of the system as well. Furthermore, since nano-satellite communication channels impose severe constraints on the volume of data that can be transferred, maximizing the software modularity should be deemed essential when designing in-orbit firmware update methods. A common update methodology is the application of patches to the software binaries. Usually, patches consist of new program data appended to the existing binary and accessed through *jump* instructions injected in specific program locations. Although this method can reduce the uploaded volume of data, its complexity is critical and may imply longer development and test times. Systems based on dynamically-linked modules could ease this procedure by allowing the replacement of the whole module (both in-orbit or during integration stages).

PolySat's design philosophy actually followed this approach, resulting in all the basic software features being implemented in shared libraries. Linux-based operating systems include the Dynamic Linker, a component that loads and links shared libraries needed by the executables at run-time. Although updating a software architecture in-orbit could be accomplished by securely replacing programs and library binaries through telemetry commands, the most valuable advantage is the ability to upgrade a component without recompiling nor modifying the rest of the architecture (e.g. kernel, drivers, other components. . .)

3.6. Centralized vs. distributed approaches

Regardless of the system topology being an extrinsic characteristic to software engineers, the choice of one or another notably affects the design of the software architecture. The majority of nano-satellites are centered around a single on-board computer that controls each subsystem though low-level, interface microcontrollers. Even though this system architectures inflict a risk on the mission if the OBC presents a failure, the complexity of the system is reduced given that most computations are performed on the same device. Communicating software components within the same environment is easy and achieved through kernel services (e.g. pipes, message queues, shared memory heaps. . .) Nonetheless, distributed approaches have also flown in previous missions, such as the one presented by the AAUSAT3 (Aalborg University, Denmark).

AAUSAT3's software architecture [3] is based on a set of applications that run on top of the platform's software stack: the bootloader, the kernel and a collection of libraries (Figure 3). Although the system

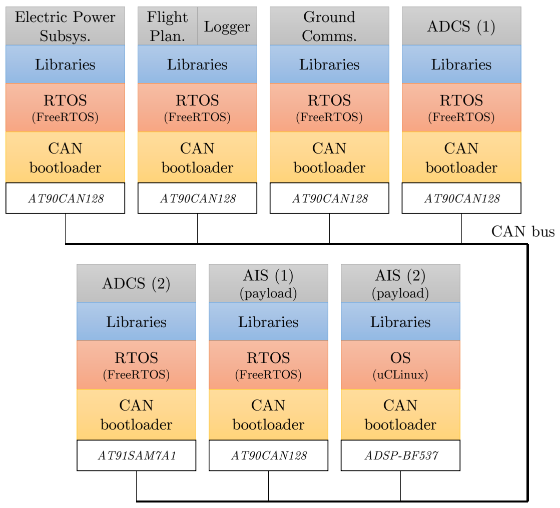

Figure 3: AAUSAT3's software platform

architecture of the AAUSAT3 did not allow to migrate all the applications to different nodes (i.e. some of them rely on hardware components which are only accessible at their node), missing one module does not necessarily imply a global mission breakdown. While isolation of components running in different nodes and application concurrency is achieved naturally and effortlessly, distributed approaches remove single-points-of-failure (SPoF) thanks to the replication of baseline components (on-board computer, OS. . .)

3.7. Software redundancy

Hardware redundancy is usually prohibitive in nano-satellites due to its cost and complexity. However, implementing redundancy at the software level is achievable and can solve some of the effects caused by SEU. Two possible types of software redundancy are envisaged and are listed as follows:

a) *Data redundancy:* the authors of [31] state that critical data may be redundantly stored within a memory device to be able to recover from SEU effects. They applied this technique for the KySat-1 flight software, an educational CubeSat project by the Kentucky Space Consortium. The KySat-1 stored three copies of sensitive data on the on-board EEPROM to ensure its integrity in the event of a bit-flip caused by radiation particles. Although this technique might not be applicable for large volumes of data, it could be a suitable solution to protect critical, non-volatile system parameters or temporary scientific data.

b) *Bootloader redundancy:* storing the boot images in Triple Modular Redundancy (TMR) memories prevents the on-board computer from starting with a corrupted image and has been implemented in some programs (e.g [28]). Despite TMR being a hardware technique, the concept of triplicating and voting a system image can also be performed in software within a single memory chip. Albeit less robust, the improvement can be adopted with little extra cost, mainly in terms of complexity, in many nano-satellite programs. The AAUSAT3 can easily illustrate the idea. Engineers of the AAUSAT3 program designed a CAN-based bootloader that can start any given application on the spacecraft boards [3]. This bootstrap system, which was designed in conjunction with the so-called Software Image Server (SWIS), was intended to perform firmware updates securely (i.e. if an application fails, the system can return to the previous version or switch to a different one). Notwithstanding, the SWIS is essentially a system that can store redundant copies of a boot image and reliably select and correct corrupted ones. Similarly, on-board computers with sufficient capacity in their ROM devices could implement simpler concepts to protect the most critical data: the kernel image.

3.8. Other techniques towards robust software

The literature covers plenty of software techniques to achieve robust software which are suitable in nano-satellite developments. Although describing them all is not the object of this paper, this section concludes with three valuable design concepts that may mitigate or help to detect problems in small spacecraft platforms.

1. *Robust communications:* exchanging information between two entities is often critical. Processes and modules may communicate to send system commands and their responses, system variables or sensitive data (e.g. subsystem configuration parameters). In some cases, this information exchange may be performed over unreliable communication channels or may be affected by SEU effects. In order to prevent unreliable delivery of digital data, the communication protocols should implement Error Detection and Correction techniques (EDAC). While the list of available EDAC techniques that can be implemented in software is extensive and can be complex (e.g. [32]) there are simple techniques that can be generally adopted with ease:

    (a) *Data integrity checks:* checksums or cyclic redundancy checks (CRC) ensure that the transferred information has not suffered any modification.

    (b) *Acknowledgments:* both positive and negative ACK segments are sent to signal the correct reception of a packet.

    (c) *Handshakes:* communication is only started after ensuring that both ends are prepared.

    (d) *Timeouts:* control the time between queries and replies to avoid hanging on deadlocked processes.

2. *Hardware and software watchdogs:* watchdogs are timers that cause automatic resets if the system under control is not responding [33]. Hardware watchdogs are present in many devices (i.e. microcontrollers) and their use may prevent global failures since they can restart the device when it is locked in unexpected deadlocks or failures. At the same time, software-based watchdogs can also be implemented. Process heartbeats can be utilized to detect deadlocks and to reset the processes whenever they occur.

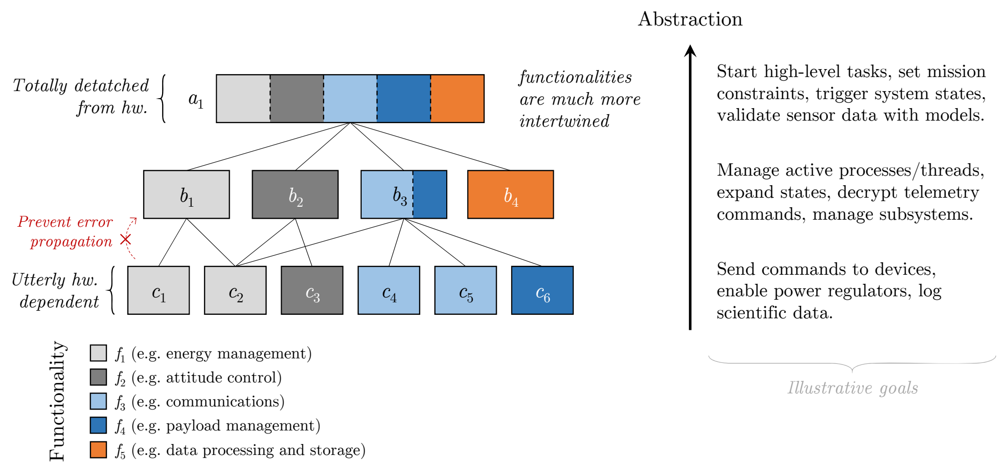
Abstraction
Totally detatched from hw.
a_1
functionalities are much more intertwined
b_1
b_2
b_3
b_4
Prevent error propagation
Utterly hw. dependent
c_1
c_2
c_3
c_4
c_5
c_6
Start high-level tasks, set mission constraints, trigger system states, validate sensor data with models.
Manage active processes/threads, expand states, decrypt telemetry commands, manage subsystems.
Send commands to devices, enable power regulators, log scientific data.
Illustrative goals
Functionality
f_1 (e.g. energy management)
f_2 (e.g. attitude control)
f_3 (e.g. communications)
f_4 (e.g. payload management)
f_5 (e.g. data processing and storage)

Figure 4: Encapsulation of functionalities in abstraction levels, or layers.

3. *Robust programming:* establishing strict coding rules and standards is critical for the management of a project with many collaborators. Stating rules that forbid certain programming constructs in order to ensure security and reliability can be key in applications where failure rate must be kept to the minimum. While the adoption of industrial standards is always the preferred alternative, their complexity and lack of knowledge can be an obstacle to many educational nano-satellite programs. Nonetheless, simpler alternatives like the ones suggested by NASA/JPL Laboratory for Reliable Software in [34] can be applied with less effort and are highly recommended.

4. Structured design criteria for nano-satellite flight software

Considering the presented compendium of techniques and concepts as a fundamental staring point, this section now poses three structured design rules that can be applied vertically throughout nano-satellite software architectures. These criteria involve both structural and functional artifacts that are complementary to the summarized techniques and which are specifically oriented to improve, even further, the groups of requirements presented in Section 2.

While most of the practices gathered in the previous section could improve the system's modularity and cope with the inherent presence of failures, they also show that these questions can be solved under many perspectives and at many levels: from low-level procedures or implementation recommendations to structural approaches and system design methodologies. Likewise, this section tries to contribute to the list of software design practices by proposing a set of guidelines that mimic some generic design rules while considering the specific needs and functional commonalities of nano-satellite flight software (i.e. mainly for Earth observation and technology demonstration missions).

4.1. Robustness through hierarchy

The architectures introduced above exhibited efforts towards the system robustness; most of them were focused on accurately detecting and minimizing the effects of errors. However, none of the practices boosted the robustness from a purely architectural perspective. Instead, they provided measures to counteract the problems: disable modules, trigger contingency modes, etc. Despite these measures being absolutely necessary, robustness can also be enriched in an abstract and generic manner through the ordering of software components. In this regard, the first proposed guideline is based on two fundamental concepts: encapsulation and goal-oriented decomposition of functionalities.

Component encapsulation is, actually, the basis of any design and its correctness not only will affect the performance of the system but can also worsen some other qualities (e.g. testability, modularity). While most nano-satellite architectures simply juxtapose modules encapsulated by the functionality of the spacecraft's subsystems, this approach complicates component interactions and lacks system perspective.

Conversely, software modules can be organized to keep hierarchical relationships. Just like organizations are divided into strata with different responsibility levels, a software architecture can also be split into several levels of abstraction (i.e. layers) in order to model the system. Each of these levels of abstraction requires the interaction with the adjacent ones to be able to develop a global function, hence establishing a hierarchical relationship. The first benefit of such modeling approach is the ability to remove error propagation paths. Layered structures where modules maintain hierarchical relationships may cut the propagation of errors if modules in each layer are sufficiently isolated. Figure 4 illustrates this idea by representing an arbitrary architecture divided into three levels of abstraction. The lines that connect each box represent module inter-dependency (i.e. "use cases") and reflect the hierarchical relationship explained above. If the modules were implemented as sand-boxed processes, these relations would be communication channels through which one process can invoke routines on another. Robust inter-process communication could allow processes in layer B to be isolated from errors in processes of layer C (e.g. a segmentation fault on module c_1 would not affect module b_1). Therefore, it becomes critical that modules which maintain some kind of dependence with others be provided with deterministic response to errors when external invocations fail unexpectedly.

In addition to that, this very vertical encapsulation of components, or "layering", can be naturally combined with the commonly implemented horizontal fragmentation based on functionality. This introduces the latter concept: goal-oriented decomposition of functionalities. While modeling the software into different levels of abstraction allows to cut error propagation paths easily, disseminating functionalities (f_i) among the layers also minimizes the complexity of each component. In accordance to this idea, a given functionality would be split into multiple tangible actions, or "goals", more or less abstract. Figure 4 illustrates possible goals with different abstraction degrees. At the same time, the figure shows that high-level components are functionally complex and may encompass several functionalities as the abstraction increases.

With this decomposition approach, components that rely on hardware (i.e. modules that control subsystems or interface with payloads) can be completely isolated from system-wide controllers that operate at a much higher abstraction level and that are critical to the mission management. Since high-level modules are untied to subsystem failures and implement abstract functions (e.g. Finite State Machines), they become easier to implement and simpler to verify by static source code analyzers. The hardware detachment presented by higher modules may enable them not only to be designed simpler but to be implemented in protected hard-real-time environments[13], inherently improving their robustness by allowing a deterministic scheduling policy of those components.

4.2. Payload-oriented modularity

The ability of a software architecture to be extended and modified relies on its modularization. Generally, identifying the subsets which maximize internal coupling and minimize coupling between modules is a demanding intellectual exercise and is influenced by subjectivity. Nonetheless, a certain lack of generality is usually needed in all engineering areas in order to deliver products that are tailored to the actual requirements and context. In this respect, software architects can follow the steps in [35] which state how to identify changeable parts from an engineering perspective, namely:

1. *Identification of the* [system] *items that are likely to change.*
2. *Location of the specialized components in separate modules.*
3. *Design inter-module interfaces that are insensitive to the anticipated changes, preventing the changeable aspects to be revealed by the interface.*

This encapsulation approach, based on the very definition of inter-module interfaces, can be embraced by nano-satellite software designers to hid the changeable parts of their products and produce replaceable modules. Besides minor modifications to correct or simplify parts of the software, nano-satellite architectures are subject to changes in parts related to their subsystems and payloads. Changes in the subsystems do not necessarily imply the removal of any of them; there is likely to be an Electrical Power System (EPS), an Attitude Determination and Control System (ADCS) and a Communications System. However, the actual hardware and their interfaces may dramatically change after a mission update. Similarly, the payloads hosted by the spacecraft will differ from one mission to another.

In accordance to the aforementioned context, the guideline here presented proposes an encapsulation of low-level modules based on payloads and subsystems, together with the definition of generic interfaces for these components. The interface is defined as a set of functions that can be invoked by other modules in

[13]They shall not access I/O devices nor perform any kind of non-deterministic call to external subsystem routine.

Table 1: Low-level modules generic interface

Function	Description
`check()`	Verify that the subsystem is ready and does not present any errors. Runs unit tests on devices, checks that there is no communication or power issue and if there are some, reports them with details.
`init()`	Configures internal parameters or the underlying devices in order to begin the execution of the *Running* routines. Acquires static resources (e.g. memory, databases, digital buses) and ensures that the system does not accumulate any previous error.
`run()`	Executes the routines that control the subsystem or payload. After an invocation of this function, the module can generate and process data, enable actuators and can communicate with other modules.
`halt()`	Reset all variables and devices, release resources and remove itself from the active list of modules (e.g. exit the process).
`<`*osf*`>()`	Interrupt the main routine or spawn a secondary execution thread to handle a custom request. Requests may trigger sub-state transitions, perform one-time actions or request instantaneous data.

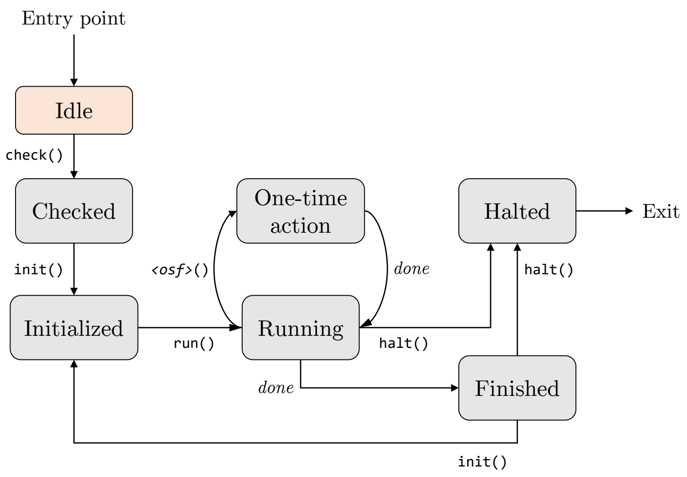

Figure 5: Low-level modules state transition network

the architecture and which modify the internal state of the component (Figure 5). Four basic functions are defined, namely, `check()`, `init()`, `run()` and `halt()`. Their implementations should account for the functionality described in Table 1. This basic API, which has some resemblance with Linux drivers management, allows the system to start and stop modules in a controlled manner. The modules could perform a set of initial checks before any other action is performed in order to guarantee that the underlying hardware or subsystems are operative. When these tests succeed, the module remains in the *Checked* state until the `init()` function is invoked. A module can, then, transition to the *Running* state after it has been correctly initialized (functions `init()` and `run()`). Most self-contained modules should be able to operate the payload or subsystem autonomously and jump to the *Finished* state once the *Running* routines are completed. Other modules, however, may never finish because they control subsystems or devices which are always active.

In addition, some modules may not be able to autonomously control the subsystem and may require external triggers to transition to internal sub-states. The so-called One-Shot Functions try to account for these triggering requirements. If the sub-states of a module were controlled by hierarchically higher components, the designers could implement custom OSF to handle those state transitions. Moreover, modules which are able to generate instantaneous data that is relevant to the rest of the architecture, could also implement specific functions to retrieve it (e.g. sensor readings, externally visible subsystem variables. . .)

Finally, either in *Finalized* or during *Running*, the function `halt()` can be called to cleanly terminate the process and release the resources (e.g. memory regions, kernel services, open databases or files, peripherals, etc.)

With this minimal, generic interface encapsulating routines that are close to the payload and subsystem hardware, changeable modules can be integrated seamlessly in an architecture. Provided that the interface is kept the same, replacing low-level components should be transparent to higher-level components and should provide the required flexibility in future nano-satellite generations.

4.3. On-board planning capabilities

Providing autonomous mission planning capabilities is, as a matter of fact, a very common approach towards autonomous spacecraft. Initially proposed by NASA for the DS-1 Remote Agent eXperiment [36], and adopted in spacecraft developments since then, the concept is fundamentally based on the definition of a set of high-level components that conform the so-called autonomy system. These modules provide the ability to both intelligently *plan* and robustly *execute* a list of timed activities based on mission goals (either self-generated or defined by ground operators), deterministic environmental conditions (e.g. orbit trajectory) and system constraints (e.g. battery state-of-charge). This design guideline proposes the functionality of an autonomy system to be included in new nano-satellite developments and simplifies its dissemination into the three elementary components shown in Figure 6.

On the one hand, a Task Planner module collects mission requirements in the form of abstract tasks.

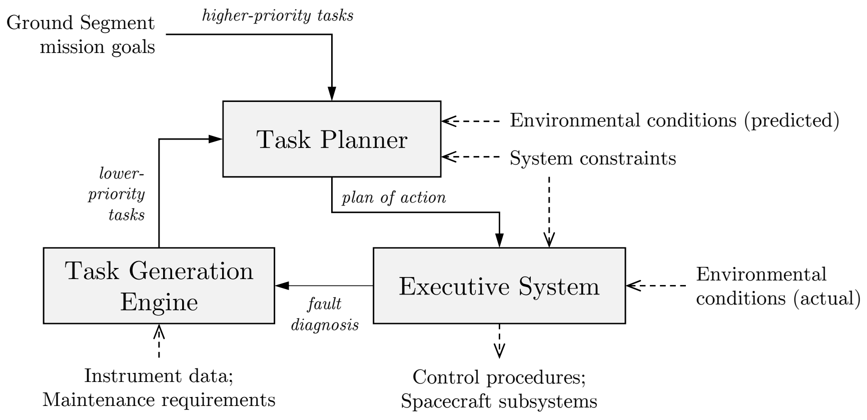
Ground Segment mission goals
higher-priority tasks
Task Planner
Environmental conditions (predicted)
System constraints
lower-priority tasks
plan of action
Task Generation Engine
fault diagnosis
Executive System
Environmental conditions (actual)
Instrument data; Maintenance requirements
Control procedures; Spacecraft subsystems

Figure 6: Autonomy System components

These tasks can be defined at the beginning of the mission, can be uploaded or modified during the satellite lifespan or can be autonomously generated by the autonomy system itself. High-level tasks may encompass a priority level which allows to weight the importance of each task. Tasks uploaded by the ground segment will tend to be prioritized over those autonomously spawned by the system. Similarly, maintenance requirements (e.g. desaturate reaction wheels, database maintenance) will likely have lower priorities to prevent them from interfering with instrument activities. In conjunction with the Task Planner, a minimal autonomy system should also encompass a robust Executive System that is able to decode the plan of action and perform all the required procedures to achieve it. Both the Task Planner and the Executive System should be consistent with the environmental conditions and system constraints. If an unexpected situation would occur or a constraint would be violated, the Executive System would cancel any related routines, activate the system safe-mode and generate a failure diagnosis report. Finally, complementing the two essential components a Task Generation Engine could be included. This optional module shall be capable to propose tasks to the system: (a) either because the previous plan has been aborted; (b) due to maintenance requests; or (c) as a result of some external observation (i.e. instrument data analysis).

Ultimately, it is worth mentioning the computational burden that an autonomy system inflicts on the OBC. Scheduling tasks and comprehensively managing their execution is an onerous endeavor and may dramatically increase the usage of computational and system resources (e.g. CPU time, memory, power). This is specially the case of deliberative task planners, where the computation required to find the optimal schedule for a finite time window can be high. Continually correcting the plan of action with up-to-date execution details and data analysis augments the autonomous capabilities of a spacecraft but may not be feasible in all cases. Because of that, nano-satellite developers may be inclined to design autonomy systems which are deployed in its wholeness at specific periods of time, generating plans of action that are not re-planned until the last scheduling window is completed or which are reactive instead of deliberative.

5. Applying design criteria

Having presented three techniques which enhance the previously justified software qualities and functionalities, this section describes an actual software architecture which applies these criteria and embraces many of the concepts presented in Section 3. This architecture is the main controlling software on-board the CubeCAT-1 nano-satellite (^{3}Cat-1), developed at the Nano-Satellite and Payload Laboratory (NanoSat Lab) of the Technical University of Catalonia (UPC BarcelonaTech). The ^{3}Cat-1 is essentially a technology demonstration mission that integrates seven different payloads within a 1U form factor [37]. The design of its flight software has targeted modularity and re-usability in order to become the precursor for future nano-satellite missions at the NanoSat Lab. Moreover, its architecture is oriented to the exploration of autonomous operations and encompasses a task planner and robust executive system as the controlling core for the satellite functionalities.

The ^{3}Cat-1's flight software architecture, depicted in Figure 7, is organized in four hierarchical layers in accordance to the first design guideline, namely, *System Core*, *Process Manager*, *System Data Bus* and *Hardware-dependent Modules*. Each of these layers has been implemented as a set of isolated processes (in some cases multi-threaded) or real-time tasks. The architecture is executed on top of a soft-real-time OS composed of a Linux kernel and a Xenomai hypervisor. This allows for an extra level of hierarchy, given that real-time tasks are scheduled with higher priority than regular Linux processes. As a matter of fact, the entire Linux kernel is run during the idle states of the RTOS' scheduler. This guarantees that critical processes such as those that control the system state, monitor health variables and control each of the underlying layers never suffer from CPU starvation nor priority inversions. The Xenomai framework provides several kernel-managed services to communicate real-time tasks with non-real-time processes. These services are used along with standard methods (e.g. named pipes, shared memory regions, signals) to provide communication interfaces between the components of the architecture.

5.1. System Core

The highest level of abstraction is set at the System Core (or *Syscore*, for short), which is composed of five different modules. The *System Safety and State Control* (*SS/SC*) module is entirely designed as a real-time component. It encompasses five Xenomai tasks which control the state of the spacecraft at the functional and energy levels. Functional states relate to system-wide actions that are taken by the spacecraft to fulfil its mission goals (e.g. perform scientific experiments, enable certain functionalities such as attitude control, establish a link with the ground segment). These functional states are coupled and induced by the energy reservoir of the spacecraft, which is also monitored at this architectural layer. The battery state-of-charge (SoC) is a critical variable provided by the *EPS* low-level module. Implementing *SS/SC* in the real-time environment was a strong design requirement because this provided greater isolation between components.

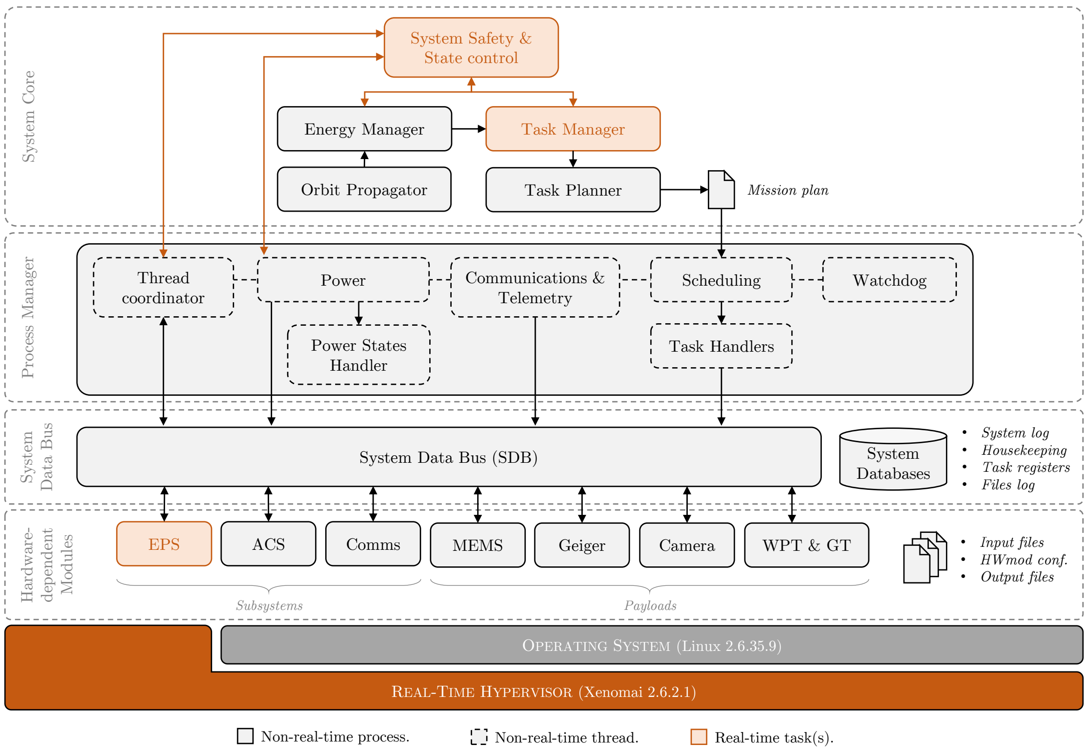

Figure 7: ^{3}Cat-1's Flight Software architecture

While, teal-time tasks are managed by the real-time kernel and hypervisor, user-space Linux processes are scheduled by the low-priority non-real-time kernel. Thus, execution of *SS/SC* is guaranteed, even when the non-real-time Linux kernel is locked in unexpected states. This design choice, forced the implementation of some *EPS* components to also be implemented as RT tasks, although their timing and latency are not critical if kept within reasonable values. For the *EPS* task, this range of allowed latency corresponded to a fraction of its period of execution, which is of 60 seconds. This range was chosen together with the hardware watchdog of the EPS microcontroller, which will cause a reset if it is locked for 16 seconds. With the *EPS* component providing critical data (i.e. SoC) to the *SS/SC*, a failure in this software component or a delayed sending of data causes a soft-reset of the on-board computer along with the corresponding logging activity to notify ground operators of this event.

Along with this real-time module, the *Syscore* includes part of the spacecraft's autonomy system: a task planner. The *Task Planner* is essentially a high-level system scheduler which generates a list of time-tagged activities for a given time window. Its behaviour is purely deliberative (i.e. *off-line*), it executes the scheduling algorithm for a finite period of time until a single solution is produced. The internal algorithm, implemented in Prolog, takes task descriptions and scheduling characteristics defined at design-time and determines when they need to be executed, within the future scheduling window. The state of the system resources are predicted either by the *Task Planner* or by the *Energy Manager*. This latter component includes models of the system that, together with the orbit propagation, allow to predict the necessary resource profiles for the Task Planner algorithm. The number of resources controlled and allocated by the *Task Planner* is limited to four, namely: energy, instantaneous power, storage capacity and operations simultaneity.

5.2. Process Manager

The second component of the autonomy system is a multi-threaded process named *Process Manager* (or *Procman*). This non-real-time component implements the Executive System and decomposes high-level commands and autonomously-generated mission plans into low-level instructions and modes of operation. Note that the Autonomy System of the ^{3}Cat-1 lacks a Task Generation Engine because none of the payloads or experiments required this functionality nor were there maintenance tasks to be scheduled autonomously.

Most of the high-level states are mapped in *Procman*'s finite state machines in order to set-up, monitor and control the low-level processes of this architecture. It is also at this architectural level where the TT&C system is implemented and all the packets are processed and generated. Thus, ground operators are capable to override *Syscore* states and have access to low-level components easily.

At the same time, power modes are managed by the *Power* and *Power States Handler* (*PSH*) components. These threads interpret energy-related commands and enable or disable spacecraft functionalities. They also have the ability to control the power mode of the OBC, allowing them to set low-power consumption modes when the energy is critical or the system is in idle states.

Finally, the *Scheduling* thread, which starts by *Syscore* request, parses the mission plan generated by the *Task Planner* and spawns a single thread to handle each of the planned activities (*Task Handlers*). These handlers initialize dedicated, low-level processes and prepare the hardware to be able to start their endeavors (e.g. enable DC/DC converters).

5.3. System Data Bus

Although the actions performed by the *Procman* involve access to the spacecraft subsystems (i.e. hardware), they are never performed directly by this process. Instead, the *Procman* can issue a set of low-level commands. This set of commands, which are still encapsulating several instructions or procedures, can be mission-specific in some cases (e.g. One-Shot Functions), but are usually generic in order to allow the *Procman* to transparently interface with whatever low-level modules the architecture has. The interface with which the low-level modules and *Procman* are connected is the *System Data Bus* (*SDB*). This layer acts as a command forwarder, delivering requests and replies from any module connected to it. It provides a second level of isolation apart from protected memory regions, given that the flow of all requests are always controlled by the *SDB* process. The *SDB* implements a simple transport protocol with timeouts which, despite not performing data integrity checks, does acknowledge the sender when a single command is executed/read by the receiver. At the same time, the SDB protocol defines restricted commands which are only available to some modules (e.g. low-level modules can not request DC/DC converters to be enabled because this action is strictly solely restricted for the *Procman*; however, all modules can request sensor measurements.)

5.4. Hardware-dependent Modules

The lower-level functionality is located in the *Hardware-dependent Modules* layer. This part of the software is composed of specialized modules (or *HWmod*'s) that can be analogous to device drivers. All *HWmod*'s implement the previously mentioned generic interface (`check()`, `init()`, `run()` and `halt()`) and, in some cases, custom OSF. *HWmod*'s execute device-level instructions like R/W operations and digital pin control, and implement communication protocols for each of the external devices connected through digital buses (e.g. UART, I^2C, SPI). They are, indeed, subsystem and payloads controllers and because of that, there is a single *HWmod* for each payload or subsystem. While the details for each of the payloads and subsystems can be read in [37, 38], it is important to note that these software components are not always running. With the exception of *EPS*, which is constantly monitoring battery SoC and PV panel's power, the rest of the *HWmod*'s are launched by demand when some of the payload or subsystem activities have to be performed.

5.5. Interfaces, files and databases

A critical aspect of this hierarchical design is to be able to preclude error propagation from one module to the other. Despite the fact that all modules run with virtually-separated memory areas, there still are inter-process or inter-task communication methods. Most of the communication channels employed in this architecture are *named-pipes* (i.e. FIFO's) where the process always involves a timeout. If a given subsystem fails, bottom-up error propagation is precluded by catching the error in the upper layer (including, for instance, timeouts) and by restarting whatever procedure or component presented the failure.

```
HW_TIMEOUT=1000000  # Microseconds
TIMEOUT_HANDSHAKE=2
TIMEOUT_SDS=5
TIMEOUT_EDS=5
TIMEOUT_ACK=7
ADAPTIVE_MODIFIER=1.0
HS_REPEAT=6
HSACK_REPEAT=10
ACK_REPEAT=5
LAST_REPEAT=3
SYNC_REPEAT=2
```

```
PROTOCOL_DELAY=1000 # Microseconds
SYNC_DELAY=100000   # Microseconds
FREQ2_RX=10         # Uplink freq.
FREQ1_RX=9D
FREQ0_RX=89
FREQ2_TX=10         # Downlink freq.
FREQ1_TX=D1
FREQ0_TX=3B
DRATE=F8            # Data-rate
SYNC1=D2
SYNC0=59
ADDR=00
FEC_DIS=0           # FEC enabled
```

Listing 1: Comms. HWmod configuration file

On the other hand, data generated by the system can also be stored in the file system, either as a regular file or in SQLite databases. The architecture encompasses up to four different databases where processes can register new files and change their state, update the state of a task and read or write system logs and housekeeping measurements (e.g. voltage, temperature, attitude states, etc.) In addition, most components of the architecture can be dynamically configured through configuration files stored in the file system. These files, which usually have a list of values for several configuration parameters, may allow ground operators to adjust the behavior of the system in-orbit (e.g. changing PID controller constants, changing the downlink frequency, etc.) Listing 1 above, shows an example of the contents of one of such configuration files.

5.6. Development assets

Finally, it is worth noting that for updatability purposes, and thanks to the adoption of a Linux OS, most of the architectural features are provided through dynamically-linked libraries. These system libraries, which are loaded at runtime, implement the SDB protocol and interfaces for *HWmod*'s, provide several wrappers to access and write to the databases, and decouple some TT&C procedures and functions from the *Procman* process (therefore allowing in-orbit upgrades of the telemetry system).

6. Conclusion

Regardless of the size of the spacecraft, the extremely harsh conditions of space can lead to a myriad of system failures. Apart from the effects of ionized particles, which not only can induce catastrophic software misbehavior but can also cause severe power failures, thermal cycles and extreme temperatures may deteriorate batteries and other components. This is specially true for missions that integrate several COTS modules, which hardly include any rad-hard or space-qualified component. Despite their reduced cost (and risk), nano-satellite missions still demand significant efforts in order to improve their robustness. At the same time, one common idea within the small spacecraft paradigm is the development of reusable platforms that can accommodate to several mission functional requirements. This approach, often crystallized into payload-agnostic commercial products (e.g. Endeavor Platform[14], by Tyvak Inc.), has expanded the capacity limits of CubeSat missions and has fostered the exploration of modular and flexible architectures. Noteworthy, low-cost and highly-modular designs have also turned nano-satellites into suitable platforms to deploy complex satellite systems such as large constellations involving thousands of units. In alignment with this idea, nano-satellites contributing to a cooperative mission potentially require their autonomous capabilities to be enhanced, not only to enable such distributed architectures but specially to ameliorate their *recoverability* and cope with their restricted operability.

The work described in this paper has introduced these three essential requirements, namely: (a) robustness; (b) payload-oriented modularity and (c) autonomy, and has mapped them to three software design guidelines: (a) robustness through hierarchical decomposition of system functionalities; (b) payload-oriented modularity; and (c) on-board planning capabilities. The presented guidelines showed ways to improve these system qualities and functionalities from the software perspective and at an architectural level. In that sense, the first two guidelines proposed a way to structure and design software modules which is fundamentally based on isolation of components and minimization of internal complexity, and encapsulation of mission-dependent subsystems with a general purpose interface. Software architectures that apply those concepts can easily replace, include or remove payload functionality and may present an improved robustness if propagation of low-level errors (i.e. those related with hardware and devices external to the on-board computer) is prevented or reliably handled. On the other hand, such hierarchical ordering of components allows system critical modules to be implemented in higher levels of the architecture and detached from hardware. Finally, this paper has proposed the design of a primitive autonomy system which is aimed at providing on-board planning capabilities to the nano-satellite. The generic autonomy system, elaborated from previous mission concepts (e.g. RAX, EO-1), is described as a set of components with a delimited goal.

The derived design criteria has been ultimately illustrated in an actual software architecture for a nano-satellite mission, the ^{3}Cat-1. Its software architecture has applied hierarchical ordering of components and payload-oriented modularization and presents a secure and reliable message-passing interface that transparently connects low-level modules with the autonomy system of the spacecraft. Moreover, the flight software is equipped with an autonomy system consisting of a multi-threaded flight executive and a fully-elastic task planner that is able to allocate more than one system resource to each activity while generating a plan of action for the spacecraft.

[14]http://www.tyvak.com/platform

References

[1] J. Bowmeester, J. Guo, Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology, Acta Astronautica 67 (6–7) (2010) 854–862. doi:10.1016/j.actaastro.2010.06.004.

[2] C. Kitts, et al., Flight results from the GeneSat-1 biological microsatellite mission, in: Proceedings of the 21st AIAA/USU Conference on Small Satellites, 2007.

[3] J. Bønding, K. F. Jensen, M. Pessans-Goyheneix, M. B. Tychsen, K. Vinther, Software Framework for Reconfigurable Distributed System on AAUSAT3, Tech. rep., Aalborg University (Dec. 2008).

[4] C. R. Boshuizen, J. Mason, P. Klupar, S. Spanhake, Results from the Planet Labs Flock constellation, in: Proceedings of the 28th Annual AIAA/USU Conference on Small Satellites, 2014, pp. 1–8.

[5] C. Ruf, S. Gleason, Z. Jeleak, S. Katzberg, A. Ridley, R. Rose, J. Scherrer, V. Zavorotny, The NASA EV-2 Cyclone Global Navigation Satellite System (CYGNSS) mission, IEEE Aerospace Conference (2013) 1–7doi:10.1109/AERO.2013.6497202.

[6] P. Ehrenfreund, et al., The O/OREOS mission – Astrobiology in low Earth orbit, Acta Astronautica 93 (2014) 501 – 508. doi:10.1016/j.actaastro.2012.09.009.

[7] F. Vuolo, N. Neugebauer, S. F. Bolognesi, C. Atzberger, G. D'Urso, Estimation of Leaf Area Index Using DEIMOS-1 Data: Application and Transferability of a Semi-Empirical Relationship between two Agricultural Areas, Remote Sensing 5 (3) (2013) 1274–1291. doi:10.3390/rs5031274.

[8] D. J. Barnhart, T. Vladimirova, M. N. Sweeting, Very-small-satellite design for distributed space missions, Journal of Spacecraft and Rockets 44 (6) (2007) 1294–1306.

[9] D. Selva, D. Krejci, A survey and assessment of the capabilities of Cubesats for Earth observation, Acta Astronautica 74 (2012) 50–68.

[10] IEEE, IEEE Std 1061-1992: IEEE Standard for a Software Quality Metrics Methodology, Institute of Electrical and Electronics Engineers, 1993. doi:10.1109/IEEESTD.1993.115124.

[11] IEEE, IEEE Std 610.12-1990: Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers, 1990. doi:10.1109/IEEESTD.1990.101064.

[12] IEEE/EIA, IEEE/EIA 12207.0-1996 Standard Industry Implementation of International Standard ISO/IEC 12207 Standard for Information Technology Software Life Cycle Processes, Institute of Electrical and Electronics Engineers, 1998. doi:10.1109/IEEESTD.1998.88083.

[13] ISO, ISO/IEC 9126: Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use, International Standard Organization, 1991.

[14] IEEE, IEEE Std 1232-1995: Trial-Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture, Institute of Electrical and Electronics Engineers, 1995.

[15] P. Clements, R. Kazman, M. Klein, Evaluating software architectures, Addison-Wesley Reading, 2001.

[16] J. L. Anderson, Autonomous Satellite Operations for CubeSat Satellites, Master's thesis, California Polytechnic State University, San Luis Obispo (Mar. 2010).

[17] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, M. Bartholomew, Verifying architectural design rules of the flight software product line, in: Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University, 2009, pp. 161–170.

[18] A. P. Cudmore, G. Crum, S. Sheikh, J. Marshall, Big Software for SmallSats: Adapting cFS to CubeSat Missions, in: 29thAnnual AIAA/USU Conference on Small Satellites, 2015, pp. 1–16.

[19] S. Chien, R. Sherwood, D. Tran, B. Cichy, et al., Using autonomy flight software to improve science return on Earth Observing One, AIAA Journal of Aerospace Computing, Information and Communication 2 (2005) 196–216.

[20] S. Chien, J. Doubleday, K. Ortega, D. Tran, J. Bellardo, A. Williams, J. Piug-Suari, G. Crum, T. Flatley, Onboard auton-

omy and ground operations automation for the Intelligent Payload Experiment (IPEX) Cubesat mission, in: Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation for Space, 2012.

[21] F. de Novaes Kucinskis, M. G. V. Ferreira, On-board satellite software architecture for the goal-based Brazilian mission operations, IEEE Aerospace and Electronic Systems Magazine 28 (8) (2013) 32–45. doi:10.1109/MAES.2013.6575409.

[22] H. Wojtkowiak, O. Balagurin, G. Fellinger, H. Kayal, ASAP: Autonomy through on-board planning, in: 6th International Conference on Recent Advances in Space Technologies (RAST), 2013, pp. 377–381. doi:10.1109/RAST.2013.6581235.

[23] J. Windsor, K. Hjortnaes, Time and Space Partitioning in spacecraft avionics, in: Third IEEE International Conference on Space Mission Challenges for Information Technology, 2009, pp. 13–20.

[24] D. Limesand, T. Whitney, J. Straub, R. Marsh, An overview of the OpenOrbiter autonomous operating software, in: Aerospace Conference, 2015 IEEE, 2015, pp. 1–12. doi:10.1109/AERO.2015.7119265.

[25] D. Evans, M. Merri, OPS-SAT: An ESA nanosatellite for accelerating innovation in satellite control, in: SpaceOps Conferences, 2014, pp. 1–11.

[26] G. Manyak, J. Bellardo, PolySat's next generation avionics design, in: IEEE Fourth International Conference on Space Mission Challenges for Information Technology (SMC-IT), 2011, pp. 69–76. doi:10.1109/SMC-IT.2011.13.

[27] M. Schmidt, K. Schilling, An extensible on-board data handling software platform for pico satellites, Acta Astronautica 63 (2008) 1299–1304.

[28] S. Tian, Z. Yin, J. Yan, X. Liu, Design and implementation of a low-cost fault-tolerant on-board computer for micro-satellite, in: Proceeding of the 7th International ICST Conference on Communications and Networking in China (CHINACOM), 2012, pp. 129–134. doi:10.1109/ChinaCom.2012.6417462.

[29] G. Manyak, Fault Tolerant and Flexible CubeSat Software Architecture, Master's thesis, California Polytechnic State University, San Luis Obispo (Jun. 2011).

[30] F. Bräuer, System architecture definition of the DelFFi Command and Data Handling Subsystem, Master's thesis, Delft University of Technology, Delft, the Nederlands (Jul. 2015).

[31] S. F. Hishmeh, T. J. Doering, J. E. Lumpp, Design of flight software for the KySat CubeSat bus, in: IEEE Aerospace conference, 2009, pp. 1–15. doi:10.1109/AERO.2009.4839646.

[32] P. P. Shirvani, N. R. Saxena, E. J. McCluskey, Software-implemented EDAC protection against SEUs, IEEE Transactions on Reliability 49 (3) (2000) 273–284. doi:10.1109/24.914544.

[33] J. Beningo, A review of watchdog architectures and their application to CubeSats, Tech. rep. (2010).

[34] G. J. Holzmann, The power of 10: rules for developing safety-critical code, Computer 39 (6) (2006) 95–99. doi:10.1109/MC.2006.212.

[35] D. L. Parnas, Designing software for ease of extension and contraction, IEEE Transactions on Software Engineering SE-5 (2) (1979) 128–138. doi:10.1109/TSE.1979.234169.

[36] E. B. Gamble, R. Simmons, The impact of autonomy technology on spacecraft software architecture: a case study, IEEE Intelligent Systems and their Applications 13 (5) (1998) 69–75. doi:10.1109/5254.722373.

[37] R. Jové-Casulleras, C. Araguz, P. Via, A. Solanellas, A. Amézaga, D. Vidal, J. F. Muñoz, M. Marí, R. Olivé, A. Saez, J. Jané, E. Bou-Balust, M. Iannazzo, S. Gorreta, P. Ortega, J. Pons-Nin, M. Domínguez, E. Alarcón, J. Ramos, A. Camps, ^{3}Cat-1 Project: a Multi-Payload Cubesat for Scientific Experiments and Technology Demonstrators, European Journal of Remote Sensing 50 (1) (2017) 125–136.

[38] S. Gorreta, J. Pons-Nin, G. López, E. Figueras, R. Jové, C. Araguz, P. Via, A. Camps, M. Dominguez-Pumar, A CubeSAT payload for in-situ monitoring of pentacene degradation due to atomic oxygen etching in LEO, Acta Astronautica 126 (2016) 456–462.