Software Development and Integration of a Hyperspectral Imaging Payload for HYPSO-1

Sivert Bakken[1], Evelyn Honoré-Livermore[2], Roger Birkeland[2], Milica Orlandić[2], Elizabeth F. Prentice[1], Joseph L. Garrett[1], Dennis D. Langer[3], Cecilia Haskins[4], and Tor A. Johansen[1]

***Abstract*— This paper presents the software architecture, development, and integration of a Commercial Off-The-Shelf (COTS) based hyperspectral imaging payload onboard the HYPerspectral Smallsat for Ocean observation (HYPSO-1) CubeSat. The chosen service-oriented software architecture provides a modular design that is planned to aid future development. The image processing onboard HYPSO-1 will be updated in-flight. We discuss here the strengths and weaknesses of our development procedures for software. The issues reported during development were analyzed and categorized, and the findings from these issues indicate the importance of early testing, code reviews, and the continuous availability of target hardware for successful software integration when relying on a modular design. A perspective of the benefits of the software architecture for a CubeSat subsystem is also given.**

I. INTRODUCTION

A. HYPSO-1 Project Overview

The HYPSO-1 CubeSat, with the Flight Model (FM) given in Fig. 1 and 2 is the first scientific satellite developed by the NTNU SmallSat Lab and is currently scheduled to be launched in December 2021. The HYPSO-1 mission deploys a 6U CubeSat with a pushbroom hyperspectral imager as primary payload [1], which captures wavelengths in the range of 400 to 800 nm, with a bandpass of 3.33 nm and a swath width of 70 km [2]. The hyperspectral images will be used to monitor spatio-temporal processes, specifically using *ocean color*.

The throughput of hyperspectral data is often limited (due to its size) by the available communication links. Thus, an On-board Processing Unit (OPU) with a Field-Programmable Gate-Array (FPGA) that allows for a modular architecture is used to process the hyperspectral image more efficiently in terms of data size reduction, power consumption, and operational time, when compared to just using a Central Processing Unit (CPU). The goal of the hyperspectral image processing pipeline is to:

- Reduce the data volume to be downloaded, both with and without loss, while retaining important spatial-spectral information,
- Deliver different data products depending on end-user needs rapidly,
- Utilize collected data from other assets [1], [3].

The payload is developed in-house, and consists of an optical telescope with a COTS camera unit, a COTS processing unit, an electronics interface board, an electrical harness, configurable software to control the payload and image processing, as well as a mechanical support structure acting as the mechanical interface to the satellite bus [2].

Next, we briefly describe the software development and integration of the payload which contains the OPU with an hyperspectral imager (HSI) and a Red-Green-Blue (RGB) imager.

B. HYPSO-1 Project Organization

By the definition used in Berthoud et. al.[4], HYPSO-1 is a university space project. Student continuity is often stated as a major challenge for university-led CubeSat projects [4], [5]. Within the HYPSO-1 project students from different BSc. and MSc. programs provide contributions to the project mainly as part of their thesis, curricular projects, and as summer interns. The team also consists of Ph.D. candidates and researchers, who partake in managing, developing, and testing and provide continuity. There is a need for project management support when developing CubeSats [4], which includes crucial documentation to transfer knowledge and extensive reviews, in addition to the development.

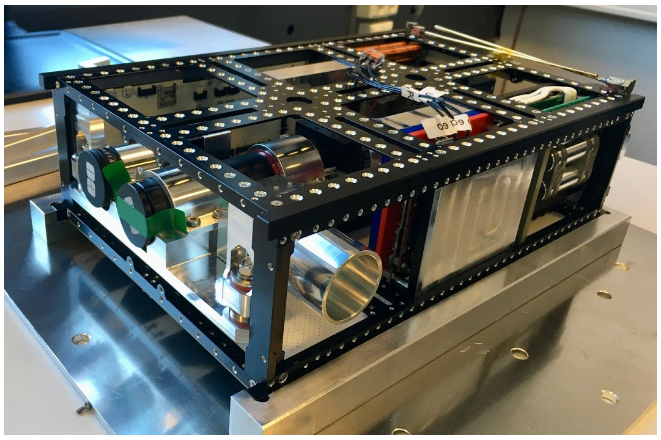
Fig. 1: Partially assembled CubeSat (NanoAvionics M6P platform) FM.

*This work is supported by the Norwegian Research Council (grant no. 270959), the Centre of Autonomous Marine Operations and Systems (NTNU AMOS, grant no.223254), the Norwegian Space Center, and the European Space Agency (PRODEX - 4000132515). NO Grants 2014 – 2021, under Project ELO-Hyp, contract no. 24/2020.

[1] Norwegian University of Science and Technology (NTNU), Department of Engineering Cybernetics
[2] NTNU, Department of Electronic Systems
[3] NTNU, Department of Marine Technology
[4] NTNU, Department of Mechanical and Industrial Engineering
Corresponding Author: `sivert.bakken@ntnu.no`

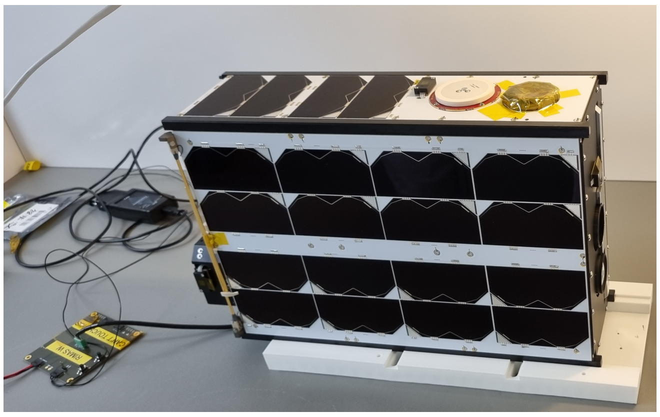
Fig. 2: Fully assembled CubeSat (NanoAvionics M6P platform) FM.

C. Scientific Software Development

Scientific software is used for the "analysis, design, testing, and deployment of software applications for scientific purposes [6, p. 1]". Software applications are developed in tandem with the research case study, algorithm development, or experiment, although some scientists may lack the necessary training in software engineering practices. This can hinder the effectiveness of the effort spent [6], [7]. Heaton et al. [7] reviewed scientific software development practices and highlighted that scientific software is often large, complex, and long-lived and may outlast the researcher — leading to problems with knowledge management caused by turnover. These challenges may be mitigated using *best practices* for software development, such as documentation [8], refactoring [8], [9], issue tracking, version control [9], peer code reviews, and design patterns.

Surveys from other university CubeSat projects recognize that software development and integration is often a challenging and time-consuming activity [4]. Reconciliation of software development with system engineering practices requires a conscious effort, and choosing and implementing the correct system lifecycle model can be challenging [10].

D. CubeSat Software Architectures

Software architectures can generally be divided into: *state-machine* types, *centralized architectures*, and *distributed architectures* using messaging systems, as described in the CubeSat flight software architecture review in [11].

State-machine type solutions offer a simplified implementation of the flight software when the functional requirements are well defined, but a change in requirements or a discovery during development can affect the states and transitions in unforeseen ways and it can be difficult to retain modularity [11]. Centralized architectures are used when there are hard real-time requirements [12], [13]. The HYPSO-1 CubeSat is constrained by the Application Programing Interface (API) available for the COTS camera, which requires an embedded Linux Operating System (OS) [13]. This OS is not designed to have real-time processing capability, and does not guarantee that certain processes are completed within a given deadline or at a specific time. Distributed architectures such as a Service-Oriented Architecture (SOA) are more flexible solutions to support an incremental development or changes in requirements, when the services are independent [11]. For SOAs, the *request-response* pattern is common when developing CubeSat Space Protocol (CSP) applications[13], and adds more flexibility and demands less coupling between modules compared with centralized architectures [11], [13]. The requirements for the HYPSO-1 software system are detailed in Tab. I. The system needs to be modular and extensible to be able to fully utilize the available contributors, and this led to the decision to use a SOA.

E. Contribution

The remainder of this paper presents the HYPSO-1 mission perspective which determined how the software architecture was designed and developed to accommodate the requirements. The corresponding management model was adapted to support the development and integration-related challenges. We tailored digital engineering practices to suit the university context [14]. In addition, we chose an architecture to enable development of the software as modules of services and features, and focused on early integration of these. This software development approach demonstrates one way of developing a CubeSat Payload with similar resources and challenges as found in the HYPSO-1 mission.

II. SOFTWARE DEVELOPMENT PROCESS FOR HYPSO-1

The mission/software development was distributed between the satellite bus manufacturer and the NTNU team. Because of this, the satellite bus manufacturer provided a remote FlatSat [4]. A FlatSat is used to mimic the hardware and software of the actual CubeSat. This remote FlatSat was used to integrate with on their premises, such that the integration and testing of the payload with the other subsystems could continue during the lockdown imposed by the covid-19 pandemic [14]. Partly as a result of the pandemic, remote software development and testing were facilitated for target hardware of the payload while it was connected with other subsystems of the 6U satellite bus through the remotely connected FlatSat. Due to the already planned infrastructure for remote integration, this eased the consequences of the pandemic for the software development, when compared to the hardware development which was more affected [14].

Specifically, multiple replicas of the OPU were connected to a FlatSat with the same electrical interfaces to be used in the Final FM of the CubeSat. This enabled development and rapid testing on target hardware in an environment that provided continuous integration. Testing is repeatedly emphasized as a high priority activity when developing CubeSats [4], as it helps reduce the time from initial code development to deployment and testing. A more detailed description of the testing setup is not given here.

TABLE I: Requirements for HYPSO-1 CubeSat software system based on [11].

Feature	Need
Extensibility	The HYPSO software system shall allow for the addition of new functionality through in-orbit upgrades.
Modularity	The HYPSO software system shall allow for separation of functionality to enable concurrent development amongst the team members.
Reusability	The HYPSO software system and modules shall be reused on different assets (such as unmanned aerial vehicles (UAVs) and multiple spacecraft).
Testability	The HYPSO software system shall enable compilation for different CPU architectures
Reliability I	The HYPSO flight model installation shall have a “golden image” with basic functionality that is redundantly distributed on the payload hardware, for recovery from software failures.
Reliability II	The system shall have watchdogs supported by the spacecraft bus in case of Single Event Effects (SEEs) or other temporary malfunctions.
Reliability III	The *boot loader* shall be protected against accidental modification.

The details on how digital engineering tools were used during software development are given in [14]. In short, the adapted agile practices have relied on a tailored Scrum approach. This was enabled using digital engineering tools provided by GitHub^{TM}. The sharing of experiences and knowledge management, coupled with a simple and well-defined workflow enabled improved verification, validation, and integration activities. However, some of the challenges typical for university-led satellite projects, specifically resource management with the presence of multiple competing objectives, were still prevalent, but less so. By using the digital engineering tools the different activities, both development and academic work, were given a common source, helping to compare and prioritize among them.

A. *Software Lifecycle*

The HYPSO mission is intended to span several years, during which multiple satellites will be deployed in a constellation. As a result the software development must continue over that period, to enable both bug-fixing and added functionality through updates in-flight. The choice of software architecture needs to enable both. The benefits of developing reusable software are significant. The basic software lifecycle from ISO 24748 [15] and the different phases for our project are shown in Fig. 3.

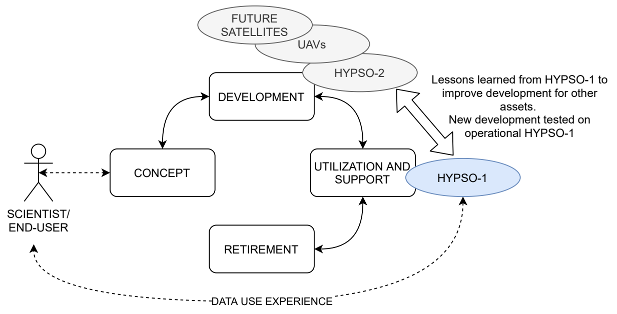

Fig. 3: Software lifecycle based on ISO 24748 with intended use shown [15].

a) Software Concept Phase: The operators and scientists provide the functionality requirements of the software system. For example, scientists expressed needs for specific validated data products that were allocated to functions such as camera parameter configurability and choice of specific `obip-services` [1]. The operators of the satellite were not involved early in the concept phase, but joined the project after the software development had begun. Their needs resulted in interface and functional changes, which were tested and refined as a part of the workflow described in [14]. The `obip-services` are intended to be updated during the mission.

b) Software Development Phase: Capstone projects were derived from the requirements found in the concept phase to support assignments to the development team [14]. These capstone projects were further divided into tasks for the Scrum sprints used to guide development. Development included implementing new features as their need became apparent, bug-fixing as errors were discovered, and refactoring when appropriate.

Through bi-weekly sprints [14], the developers needed to resolve that each identified issue or task was estimated, and goals for the following sprint were set. At the end of each sprint the progress was reviewed and a new sprint was planned for the next period.

c) Software Utilization and Support Phase: Not all incremental software improvements result in a set of services that will be used in-flight. Nevertheless, there is an underlying goal of doing rudimentary testing of each proposed code contribution so that the most recent version of the payload software is working. More extensive testing, with all available subsystems, is performed before making a flight-viable release. In this process we are utilizing semantic versioning [16], namely a standardized method for documenting changes between versions. This provides a high-level description of the changes that can be more easily communicated to other team members and software users.

d) Software Retirement Phase: It is foreseen that some modules will be reused in other systems, and the knowledge management in GitHub^{TM} can support this. With the high personnel turnover often found in CubeSat projects it is important to have good knowledge transfer to ensure progress [4], such that modules can be re-used and the same functionality is not developed multiple times. To facilitate this, the future systems are planned to utilize the same or similar COTS components, while only making incremental improvements.

B. *Payload Software Architecture*

As defined earlier, HYPSO-1 employs a SOA. This allows for services to be provided by application components to other components (both within the same subsystem, and with other subsystems) via network communication. A *service* is here defined as a set of related functionality that can be requested from the user [17].

To interface with the chosen COTS camera, the supplier's official closed source API is used, which constrains the selection of an OS to be an Embedded Linux system. However, this OS is a well-established and widely used OS, and does not come with the unknown bugs of a custom build OS, which can be common for CubeSats. The OS image of the payload is therefore built using the open-source PetaLinux build system from Xilinx to deploy an Embedded Linux system [12]. The chosen Zynq 7030 System-on-Chip (SoC) with the FPGA used for the on-board data handling is supported by PetaLinux [13]. The build system was customized to include the payload application, and necessary drivers, as applications within the OS image [12]. Wtih a backup of the OS image

The software is here defined as executable applications started on boot. These applications are the service providers of the payload, as shown in the simplified diagram of Fig. 4, and they are planned to be updated in-flight. The File Transfer (FT) service is made to interface with the satellite provider's API and Interface Control Document (ICD), and is compatible with both the payload and the rest of the satellite bus subsystems. The HSI and RGB services send commands to the respective cameras. The CSP and OS communicates with the other processes and provide Linux commands respectively. The Telemetry service logs the state parameters of the payload over time.

By separating the functionality as a stand-alone executable we can restart the subsystem into a known state at each power on. There is no persistence in the OS image, so it will experience a fresh start at each boot. In addition, the executable service provider can be hot-swapped during execution. This means that we can update the software by simply uploading a new file without altering the booting sequence. This enables us to have multiple versions of the service provider available during operations. This addresses the first requirement in Tab. I about extensibility, an uncommon feature of traditional spacecraft systems.

Firmware is defined as the system image, or `opu-system`, which consists of the kernel, device tree, root file system, and the FPGA programmable logic bitstream [12]. An updatable primary image is stored on an SD-card, and a backup is stored in embedded memory. This backup shall not be changed after FM assembly. The difference between the firmware and software update process is primarily due to the amount of data that needs to be transmitted for an update, and the level of risk associated with performing their updates. Backward error recovery for firmware updates is done by restoring the image from embedded memory, if the primary image fails to load after attempting to boot a set number of times.

The root file system is always loaded from the boot image. This ensures that the system enters a known state after boot. Payload data and different versions of software applications are stored on an SD-card, and can be taken into use at run-time. The planned updates will add new versions of the service providers [1] with new algorithms and bug fixes.

The `opu-services` application/executable is the minimum viable product. This application captures, compresses [18] and transfers hyperspectral and RGB images, communicates with other subsystems, provides telemetry information and provides remote shell access to the OPU. A version of `opu-system` and `opu-services` that passed all our tests were integrated in the payload FM and shipped to the satellite bus provider for integration with the rest of the satellite bus during the early summer of 2021. Software intended to expand the capabilities of the payload will be added during the mission lifespan.

A planned future application is the On-board Image Processing (OBIP), which will process and derive high-level low-latency data products from the captured data [1], [3], such as results from classification and target detection. The application is marked in red to indicate that it is a part of a future extension of the `opu-system` in Fig. 4. With this proposed SOA, we expect to be able to update only the software containing the services available from the payload with low risk to the mission, when compared to updating the entire system image. With this modular and extensible design code contributors can add to the system in the form of a single application or by expanding the services available from an existing application on the platform. Updating the `opu-services` application entails risks that are necessary to meet changing processing requirements.

The application repository (*hypso-sw*), in addition to deploying the payload executable known as `opu-services`, also builds the `hypso-cli`. `hypso-cli` is the Command Line Interface (CLI) used to send commands via CSP packets that will be propagated from the mission operator to the

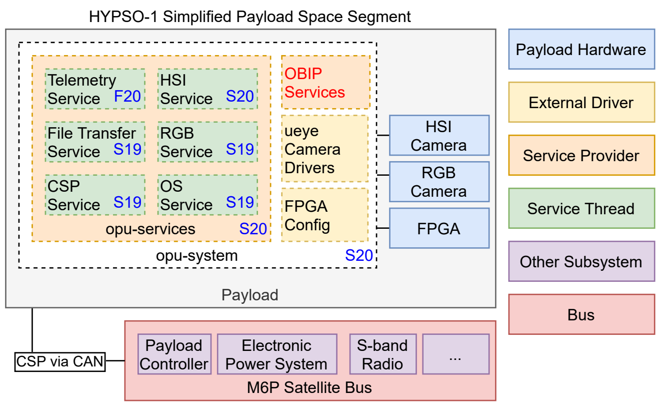

Fig. 4: Simplified overview of the HYPSO-1 payload with dashed boxes as software components and solid boxes as hardware components. The season and year for integration of the module are given as blue text.

satellite bus, and subsequently to the payload subsystem [13], as indicated by Fig. 4. CSP is a lightweight network protocol, resembling IP, that can be used as a network layer between both physical subsystems and between different software services internal to one subsystem. CSP supports several hardware layers, where HYPSO-1 relies on Controller Area Network (CAN) and Universal Synchronous and Asynchronous Receiver-Transmitter (USART). The `opu-system` starts the `opu-services` in the version packaged into the image on boot [12]. The system supports having multiple versions of `opu-services` in non-volatile memory (SD-card) and can change between them at runtime. New versions, with bug fixes, can then be added without compromising old ones.

III. Software Issue Analysis

We aim to better understand what kind of problems occurred in the development and integration process of the satellite and how we solved a subset of them. To do so, we encouraged the team to document discovered bugs, desired features, and other proposed changes as GitHub™ "issues". Then we surveyed all open and closed issues labeled as *bug* that resided in the repositories and separated these into four categories based on how they were discovered:

- When interfacing between subsystems (*Sub.*),
- When testing the internal functions of a module or interfacing between services for the payload (*Mod.*),
- When considering scientific data handling (*Sci.*),
- or miscellaneous (*Misc.*).

The categorization was done in two rounds; first independently by three of the authors, and then as a group with a clarification of interpretations of the categories. The bugs related to FT, use of external drivers with their underlying errors, and general subsystem communication from the perspective of the payload were classified as *Sub.* bugs. Issues related to the functional behavior of modules and their internal communication were classified as *Mod.* bugs. Issues related to data handling were classified as *Sci.* bugs. The issues that did not fit any of these categories, e.g., virtual build environment, were classified as *Misc.* bugs.

TABLE II: Categorized issues labeled as bugs.

Repository	# of issues	% Closed	Sub.	Mod.	Sci.	Misc.
hypso-sw	78	76.92	27	39	2	10
opu-system	22	100.00%	2	15	0	5

The findings from over the course of 2 years are given in Tab. II, and show that most of the issues reside in the `hypso-sw` repository. This is also the repository that has had the most contributors. Here, the issues are found mainly in *Sub.* and *Mod.* bugs, with the other categories being less prominent. For the *Mod.* bugs detected, the FT service is a frequent source of discovered bugs. This FT service was developed at an early stage of the project, as seen in Fig. 4, and has been invoked frequently when testing the functionality of other services as well.

The HSI functionality was developed as a standalone subroutine before being integrated as a service within `opu-services`. If the focus had been on earlier integration of the HSI service, there would probably have been fewer challenges to resolve during integration. The CSP and OS services rely on third-party implementations with larger communities, and fewer or no bugs are related to these. The RGB service is smaller/simpler and has been invoked less during testing and has fewer bugs related to it. The telemetry service is a late addition to `opu-services`, and it is not invoked by other services, and few bugs are related to this service.

Automated tests helped discover many issues. However, most of the issues reported were not related to the code under review directly. That is, the exploratory nature of manual testing made it possible to discover issues beyond the scope of what was initially supposed to be tested.

Other issues in `hypso-sw` are related to feature requests and other enhancements.

IV. Refactoring and Future Missions

In software development, refactoring is defined as "a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior." [19, p. 565] This is an enhancement activity that has been performed to make the code base more maintainable and accessible for future contributors. As an example, the HSI service has been refactored multiple times to better divide its functional parts into smaller, more maintainable functions. This has made it easier to understand without changing its behavior. By this refactoring, the performance of some routines was improved, thereby providing performance gains for the arguably most important service of the payload, and thus delivering a more capable mission.

A second satellite is planned, namely HYPSO-2 [20]. This satellite is intended to feature an additional secondary payload, a Software Defined Radio (SDR). While this payload also is based upon a similar hardware platform as the OPU, it will have its separate system image and application image. Parts of the codebase have been refactored to better support the `hypso-sw` for multiple payloads. This relates to the application services that integrate the payload with the bus, such as shell, CSP and FT services in addition to the telemetry services. The SOA and a common OS running on both payloads made this possible. Successfully supporting this development within the existing *hypso-sw* repository is a demonstration of the extensibility, modularity, and reusability of the software as specified in Tab. I.

The chosen SOA provides a high-level abstraction that enables further development of the service modules of `hypso-sw` and development of new services for future satellites. The first satellites will also benefit from future development as they can be updated. Modularization, refactoring, and generalization have been the key principles used to meet the driving needs that are defined in Tab. I [20]. Through the refactoring process, several common factors for future payload were identified to further ease future development.

V. Discussion and Conclusion

The software development process for the HYPSO-1 project provided insight into how such systems could better be developed and integrated in the future.

Relevant literature emphasizes the importance of testing [4]. Experiences from developing the HYPSO-1 software made it clear that testing and integration will help with discovering errors, as more testing of a given service uncovers more bugs. It can be challenging to find the human resources to test extensively, and this activity can be challenging to motivate in a university setting [4], [14]. Integration with other modules or subsystems is also prone to introduce new insight. The analysis of issues registered for the HYPSO-1 software given in Sec. III, with a focus on those labeled as bugs, also substantiates frontloading of testing, and early integration, as recommended by [4].

The choice of software architecture made it possible to develop functionality as independent components or services. This helped to generate several contributions from multiple contributors with high turnover, without adversely affecting the functionality or development cycle.

The separation of platform and application, Embedded Linux OS and `opu-services` respectively, was done to make it possible to perform in-orbit upgrades with lowered risk to the mission. Upgrades of the system and software have yet to be demonstrated in-orbit. However, this upgrade functionality has been tested extensively, e.g. as a part of software development itself. That is, new software contributions were regularly developed on the target hardware, and the upgrade functionality was used to deploy and test them.

The development and integration of future services for new payloads have demonstrated the perceived benefits of the chosen software architecture [20]. The SOA enables the reuse of code for future development.

These experiences will aid in the development of future satellites that are planned by the NTNU SmallSat Lab, and can be scaled to other similar systems as well.

References

[1] M. E. Grøtte, R. Birkeland, E. Honoré-Livermore, S. Bakken, J. L. Garrett, E. F. Prentice, F. Sigernes, M. Orlandić, J. T. Gravdahl, and T. A. Johansen, "Ocean Color Hyperspectral Remote Sensing With High Resolution and Low Latency–The HYPSO-1 CubeSat Mission," *IEEE Transactions on Geoscience and Remote Sensing*, pp. 1–19, 2021. DOI: 10.1109/TGRS.2021.3080175.

[2] E. F. Prentice, M. E. Grøtte, F. Sigernes, and T. A. Johansen, "Design of a hyperspectral imager using COTS optics for small satellite applications," in *International Conference on Space Optics — ICSO 2020*, B. Cugny, Z. Sodnik, and N. Karafolas, Eds., International Society for Optics and Photonics, vol. 11852, SPIE, 2021, pp. 2172–2189. DOI: 10.1117/12.2599937.

[3] J. L. Garrett, S. Bakken, E. F. Prentice, D. Langer, F. S. Leira, E. Honoré-Livermore, R. Birkeland, M. E. Grøtte, T. A. Johansen, and M. Orlandić, "Hyperspectral Image Processing Pipelines on Multiple Platforms for Coordinated Oceanographic Observation," *11th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2021.

[4] L. Berthoud, M. Swartwout, J. Cutler, D. Klumpar, J. A. Larsen, and J. D. Nielsen, "University cubesat project management for success," *Proceedings of the AIAA/USU Conference on Small Satellites*, 2019. DOI: https://digitalcommons.usu.edu/smallsat/2019/all2019/63/.

[5] J. Grande, R. Birkeland, A. Gjersvik, and C. Stausland, "Norwegian student satellite program - lessons learned," in *Proceedings of The 68th International Astronautical Congress*, 2017.

[6] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and J. C. Carver, "Software engineering practices for scientific software development: A systematic mapping study," *Journal of Systems and Software*, vol. 172, p. 110 848, 2020. DOI: 10.1016/j.jss.2020.110848.

[7] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207–219, 2015, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.07.011.

[8] Y. Li, "Reengineering a scientific software and lessons learned," in *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 41–45. DOI: 10.1145/1985782.1985789.

[9] K. S. Ackroyd, S. H. Kinder, G. R. Mant, M. C. Miller, C. A. Ramsdale, and P. C. Stephenson, "Scientific software development at a research facility,"

IEEE Software, vol. 25, no. 4, pp. 44–51, 2008. DOI: 10.1109/MS.2008.93.

[10] M. W. Maier, "System and software architecture reconciliation," *Systems Engineering*, vol. 9, no. 2, pp. 146–159, 2006.

[11] C. E. Gonzalez, C. J. Rojas, A. Bergel, and M. A. Diaz, "An architecture-tracking approach to evaluate a modular and extensible flight software for cubesat nanosatellites," *IEEE Access*, vol. 7, pp. 126 409–126 429, 2019. DOI: 10.1109/ACCESS.2019.2927931.

[12] J. A. Gjersund, "A reconfigurable fault-tolerant on-board processing system for the hypso cubesat," M.S. thesis, NTNU, 2020.

[13] M. Hov, "Design and implementation of hardware and software interfaces for a hyperspectral payload in a small," M.S. thesis, NTNU, 2019.

[14] E. Honoré-Livermore, R. Birkeland, S. Bakken, J. L. Garrett, and C. Haskins, "Digital Engineering Management in an Academic CubeSat Project," *Special Issue on Systems Engineering Challenges in Journal of Aerospace Information Systems*, 2021.

[15] International Organization for Standardization, *ISO/IEC/IEEE 24748-1:2018(E) Systems and Software engineering life cycle management*, 2018.

[16] *Semantic versioning 2.0.0 — semantic versioning*, https://semver.org/, (Accessed on 07/29/2021).

[17] M. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE2003*, 2003, pp. 3–12. DOI: 10.1109/WISE.2003.1254461.

[18] M. Orlandić, J. Fjeldtvedt, and T. A. Johansen, "A Parallel FPGA Implementation of the CCSDS-123 Compression Algorithm," *Remote Sensing*, vol. 11, no. 6, 2019, ISSN: 2072-4292. DOI: 10.3390/rs11060673.

[19] S. McConnell, *Code complete*. Pearson Education, 2004.

[20] T. O. Moxnes, "A common software framework for a cubesat with multiple payloads," M.S. thesis, NTNU, 2021.

ACKNOWLEDGMENT

The authors would like to thank all the MSc. students contributing to *opu-system* and *hypso-sw*, especially M. Danielsen, M. Hov, T. O. Moxnes, and J. A. Gjersund. We thank the numerous reviewers and are grateful for the participation of NanoAvionics and their continuous feedback throughout the project.