SSC21-XII-09

Design and Validation of an Autonomous Mission Manager towards Coordinated Multi-Spacecraft Missions

Antony Gillette, Alan George
NSF Center for Space, High-performance, and Resilient Computing (SHREC) - University of Pittsburgh
4420 Bayard Street, Suite 560, Pittsburgh, PA; 352-389-1536
agillette@pitt.edu

J. Patrick Castle
NASA Ames Research Center
M/S N269-1, Moffett Field, CA 94035; 650-604-0435
joseph.p.castle@nasa.gov

ABSTRACT

For ambitious upcoming aerospace missions, autonomy will play a crucial role in achieving complex mission goals and reducing the burden for ground operations. Standalone spacecraft can leverage autonomy concepts to optimize data collection and ensure robust operation. For spacecraft clusters, autonomy can additionally provide a feasible method of ensuring coordination through onboard peer-to-peer scheduling. However, in exchange for providing flexible mission capabilities and operational convenience, autonomy introduces additional uncertainty and software complexity, which complicates the mission assurance process. This research presents a framework for designing and testing schedules consisting of heavily constrained tasks.

The core of this framework, the Schedule Manager (SM), manages tasks by associating constraints with each task including time windows, task priority, conflict categories, and resource requirements, which assures that tasks will only run when capable. This increased control over individual tasks also

improves the modularity of the overall mission plan, and provides a built-in fail-safe in the event of unexpected task failure through the loading of predefined contingency schedules. The SM can use estimated task durations and resource requirements to simulate schedules ahead of time, which can be used on the ground for schedule validation and onboard as a method of prognostics and to calculate resource availability windows. The ability to predict availability windows onboard and dynamically adjust depending upon currently scheduled tasks enables peer-to-peer tasking and scheduling. For example, a spacecraft can schedule a coordinated action by broadcasting the task requirements in an availability window request to all applicable spacecraft. Then, based upon the availability windows received from each spacecraft, the coordinating spacecraft can then issue a final task scheduling command with a much lower probability of conflict.

The SM has been integrated with the core Flight System (cFS) from NASA, which has flight heritage on previous successful large-scale missions such as the Lunar and Dust Environment Explorer (LADEE). This integration is in the form of a cFS application called the cFS Schedule Manager (CSM), which will manage the operations for the Space Test Program Houston 7 Configurable and Autonomous Sensor Processing Research (STP-H7-CASPR) experiment that is planned for launch on SpaceX-24 to the International Space Station (ISS) in December 2021. Software validation was achieved with cFS unit tests, functional tests, and code analysis tools. Demonstrations were built using the COSMOS ground station and the 42 spacecraft simulator, and these were tested with a cluster of development boards in the loop as representative flight hardware.

INTRODUCTION

The past few decades have seen research and development into many different types of distributed spacecraft missions.[1] Ambitious mission concepts featuring constellations of satellites to achieve mission objectives previously deemed to be impossible or cost-prohibitive are a significant motivating factor for the continued improvement of autonomous spacecraft constellation management capabilities. However, there are many required components that are necessary to enable these missions,[2] and the increased complexity of these components relative

to their counterparts for solo spacecraft missions widens the gap between theoretical research (e.g., constellation navigation and consensus algorithms) and realized implementations tested on flight-like hardware. Also, due to the variety in types of multi-spacecraft missions and corresponding required capabilities, it is more challenging to leverage concepts and software developed for prior missions compared to solo spacecraft missions.

This research aims to address the current challenges associated with constellation mission software research and development through a simplified and general-purpose baseline that can demonstrate autonomous mission management concepts on resource-constrained embedded platforms. With the focus of this study being on developing software that has a path to flight, many additional design considerations are necessary, such as the programming language and paradigms used, resource requirements, and feasibility of testing to the standards required for flight software. The resulting solution was achieved in the design of the Schedule Manager (SM), a library of schedule-management functions written in C that targets many design considerations, such as simplicity, scalability, efficiency, safety, and interoperability. With the SM as the core, integration with the core Flight System (cFS) developed by NASA Goddard Space Flight Center (GSFC) was achieved through the cFS Schedule Manager (CSM) application to demonstrate performance capabilities in the context of a realistic flight software solution.

The SM enables the upload and execution of schedules consisting of heavily constrained tasks. Each individual task can be assigned a selection of optional constraint fields, and through the proper design and upload of task schedules, a user can control how the system will respond even with many unpredictable and uncertain influencing factors. By having constraints assigned to individual tasks, the mission planning phase can be modularized to minimize operational complexity. If individual tasks are properly designed with specified conditions and resources required for successful execution, then unpredictable situations such as delays in the availability of shared resources will be handled automatically. The development of the SM required iterative redesigns and tradeoff explorations to be able to achieve a satisfactory balance between the usability of the system and the support of the desired features initially conceptualized.

The first phase of research for the SM and CSM was to enable autonomous control for solo spacecraft missions, with a practical goal of enabling autonomous operations for space missions developed at the National Science Foundation (NSF) Center for Space, High-Performance, and Resilient Computing (SHREC) based at the University of Pittsburgh. At SHREC, two missions have been launched and one planned for the U.S. Department of Defense Space Test Program (STP) Houston, namely STP-H5 CHREC Space Processor (STP-H5-CSP) which launched on SpaceX-10 in 2017,[3] STP-H6 Spacecraft Supercomputing for Image and Video Processing (STP-H6-SSIVP) which launched on SpaceX-17 in 2019,[4] and STP-H7 Configurable and Autonomous Sensor Processing Research (STP-H7-CASPR) which is planned for launch on SpaceX-24 in December 2021,[5] and mounted on the International Space Station (ISS). Figure 1 illustrates STP-H5-CSP, STP-H6-SSIVP, and STP-H7-CASPR. The CASPR mission uses the CSP and two SHREC Space Processors (SSPs), based on the Zynq-7020 and Zynq-7045 systems-on-chip (SoCs), respectively. As a multi-technology demonstration, CASPR includes many subsystems that cannot be run concurrently due to power constraints and would benefit from the ability to specify start times and resource constraints for specific operations (such as for image capture).

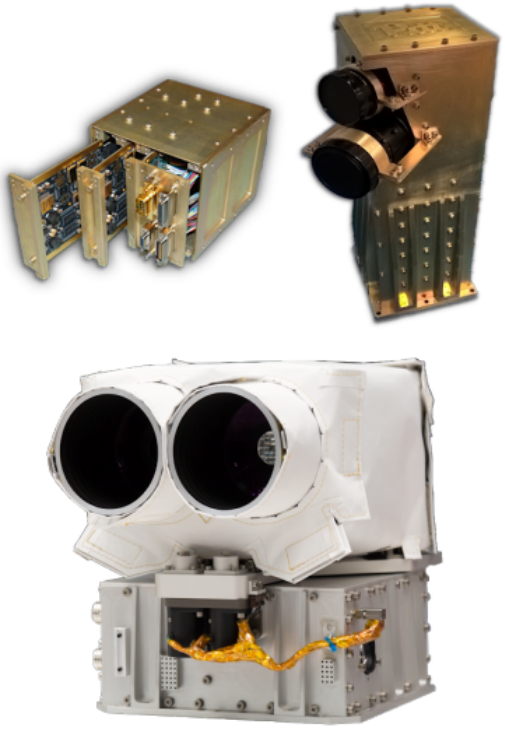

Figure 1: STP-H5-CSP (Top Left), STP-H6-SSIVP (Top Right), STP-H7-CASPR (Bottom)

The second phase of this research was to demonstrate the applicability of the SM and CSM for multi-spacecraft missions that require autonomous collaboration, as well as to emphasize the feasibility of validation and testing of the established software framework to show suitability for future high-profile missions. Towards this goal, onboard schedule simulation and resource-availability prediction capabilities were designed and added to the SM. These capabilities can be used as an additional method of schedule validation as well as to enable dynamically scheduled tasks for multi-spacecraft missions with collaborative availability communication.

In the following sections, first a background of related technologies and related works will be covered. Next, the design of the SM and CSM will be presented, followed by an overview of the methods used for testing, and finally details on the tested demonstrations.

BACKGROUND

This section will begin with an overview of the capabilities and benefits of cFS including a description of recent cFS mission examples as a reference. Next, the tools used to assist in simulation and testing will be described. Finally, this section will conclude with a summary of related research.

Capabilities of the core Flight System and Adoption Benefits

The initial goal of this research was to improve the autonomous capabilities of cFS. The cFS framework includes the core Flight Executive (cFE), which provides a set of apps that support cFS such as the Software Bus (SB) app which enables the sending of command and telemetry messages between all cFE/cFS apps via a publish-subscribe message bus with accepted packets following the Consultative Committee for Space Data Systems (CCSDS) protocol. CCSDS headers contain fields for commands including message ID, sequence number, packet length, command code, and checksum, as well as timestamp for telemetry, to support the routing of messages between apps. The cFE also includes other apps such as the Executive Services (ES) app which enables the startup and runtime management of other cFE and cFS apps, and the Table Services (TBL) app which manages a system for the validation and loading of compiled tables. Any apps developed outside of the cFE core are considered cFS apps, and NASA GSFC has released many open-source cFS apps on GitHub such as the Command Ingest (CI) app which enables the system to receive external SB command messages through a UDP socket, the Telemetry Output (TO) app which enables the output of SB telemetry messages on a UDP socket, the Schedule (SCH) app which can send defined SB command messages periodically, the Stored Commands app (SC) which can load tables of SB command messages and send them at specified times, and the Limit Checker (LC) app which can monitor SB telemetry messages and send specified SB command messages if certain thresholds are exceeded. cFS is built on top of NASA's Operating System Abstraction Layer (OSAL), enabling software deployment to multiple OS platforms, and the OSAL includes several Platform Support Packages (PSPs) to assist with hardware compatibility.[6]

The functionality provided by these cFS apps is relatively straightforward but has been used to support many high-profile missions. An example of a prior mission that used cFS is Dellingr, which was a 6U CubeSat developed by NASA GSFC.[7] Dellingr used the LC app to monitor and respond to potential issues such as unsuccessful antenna or boom deployment, invalid radio/heater configurations, and other system errors. After the detection of a system failure, the error could be handled by sending a SB command message to trigger a suitable SC schedule to perform actions such as restarting operation sequences, resetting the system, and entering safe mode. Dellingr encountered various issues on launch, but many of these issues were successfully fixable through a variety of methods.[8] Another detailed example of using cFS to manage spacecraft operations is provided by the Lunar and Dust Environment Explorer (LADEE) developed at NASA Ames Research Center (ARC).[9] In addition to using the SC and LC in a similar manner to Dellingr, LADEE also used the Health and Safety (HS) app to perform actions such as handling unresponsive tasks in a configurable manner, ranging from resetting an app a set number of times to triggering an immediate processor reboot depending on the severity of the issue. The LADEE mission design process outlined many potential faults and designed custom methods to handle each situation.[9]

The above two missions provide well-described examples of cFS usage but make up a small portion of the overall list of missions that have successfully used cFS to achieve their goals. Out of the various open-source flight software frameworks available to the space community, cFS stands out as a capable and flight-proven solution.[10] By using a commonly reused flight software solution like cFS, prior lessons learned and experience can be built upon and mis-

sion development work can more easily benefit subsequent missions in the future. The level of open-source activity for cFS on GitHub has been on the rise recently, with many discussions with the community and collaboration on the development and addition of new features. Through the adoption and usage of cFS, many practical benefits can be gained such as a reduction in required development time, a lower level of risk due to the tests and analysis performed for prior usage, and the ability to ask for support from the community when encountering software issues or contemplating the practicality of adding new features.

Simulation and Testing on Software and Hardware Platforms

A baseline cFS testing framework is provided by OpenSatKit, an open-source project combining NASA GSFC's 42 spacecraft simulator and Ball Aerospace's COSMOS ground system to provide a full end-to-end mission development workflow.[11] The 42 simulator supports features such as the dynamics of gravity, magnetic fields, and atmospheric density using configurable models, and it also provides methods to visualize and control the movement of spacecraft in orbit. The COSMOS ground system enables the manual or automated sending of predefined commands and parsing of telemetry data. By using these two tools along with cFS, a mission software developer can realistically interface with a simulated spacecraft and see how it would behave in orbit. For the testing of constellation missions, multiple targets can be added in COSMOS and 42, with the main additional necessity being the setup of virtual machines or containers to run multiple instances of cFS.

For more realistic testing, embedded development boards can be used to account and test for compatibility and capabilities of the target platform. The flight processors developed at SHREC are based on Zynq-7000 SoC devices developed by Xilinx. The Zynq-7020 and Zynq-7045 featured on the CSP and SSP, respectively, both include a dual-core ARM Cortex-A9 processor and 7-Series FPGA fabric. Many development boards are commercially available that include the same devices such as the ZedBoard and PYNQ-Z2 board, which use the Zynq-7020, and the ZC706 board, which uses the Zynq-7045. Testing on desktop computers means compiling for an x86 architecture and generally without memory or processing constraints. Cross-compiling apps for the ARM architecture and testing on development boards provides a more accurate runtime estimate and will bring attention to potential issues such as memory usage, thread congestion, and kernel compatibility. Aside from custom components such as the radiation-hardened watchdog and NAND flash memory and related tests, testing on a Zynq development board will produce results similar to testing on the CSP or SSP. As an added benefit of testing on development boards, building hardware-in-the-loop demonstrations with many nodes is much more affordable and with a more manageable form-factor as can be seen by a picture of the PYNQ-Z2 cluster testbed used for this work in Figure 2.

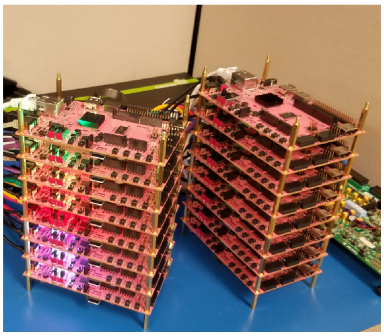

Figure 2: Cluster of 16 PYNQ-Z2 Development Boards

Related Works

A prominent early example of spacecraft autonomy was provided by the Autonomous Sciencecraft Experiment (ASE) on NASA's Earth Observing One (EO-1) spacecraft.[12] The onboard planner used on ASE, named the Continuous Activity Scheduler Planner Execution and Re-planner (CASPER), has been used on multiple missions since, such as the Intelligent Payload Experiment (IPEX) CubeSat mission.[13] The responsibilities of CASPER on IPEX included receiving plans generated on the ground and then autonomously handling conflicts due to overlapping observation goals, excessive resource consumption including CPU, RAM, storage, and power, and deviations in activity durations and image data product sizes.

Another example spacecraft autonomy solution is provided by the flight software that was developed for the CubeCAT-1 nanosatellite (^{3}Cat-1).[14] An onboard task planner design was used to schedule time-tagged activities while managing four resources: energy, instantaneous power, storage capacity, and operation simultaneity.

Regarding autonomy for distributed satellite missions, Araguz et al.[2] overviews the current state of distributed satellite systems, including potential

benefits for the application of autonomy to distributed missions, as well as guidelines for concepts to consider in the development of autonomous mission planners. From this detailed overview, it is evident that there are many benefits but also corresponding challenges associated with applying autonomy to distributed satellite systems.

APPROACH

Towards the goal of achieving autonomous coordination of spacecraft constellations, the SM and related components were designed and developed with many tradeoffs considered. One significant challenge to overcome was feature creep along with an exponentially growing level of complexity involved with the understanding and usage of the developed framework. For example, flexibility and capability can be gained by adding additional features, but at the cost of overall simplicity, predictability, and safety. Modularity of functional components can help with overall testing and development, but correspondingly introduces integration complexity and challenges.

The core component of the autonomous mission-management solution described in this paper is the SM, which manages the execution of schedules of heavily constrained tasks by defining consistent rules for how each constraint affects the execution logic of tasks or their corresponding contingencies. When used standalone, tasks for the SM can represent either external processes such as executable binaries or scripts, or alternatively function calls which are generally used for schedule management. When the SM is used in a cFS app in the form of the CSM, tasks can additionally represent raw SB command messages or cFS function calls to fit in the paradigm of cFS-based mission design. The CSM enables interfacing with cFS using standardized commanding and configuration methods. The following sections will describe the capabilities of the SM and CSM, followed by the design and application of the coordination and availability prediction capabilities added to the SM.

Schedule Manager Concept Overview

The SM enables a user to design a schedule of tasks, upload the schedule to the running SM instance in various acceptable formats, and watch the execution of the tasks in real-time. The uploaded schedule first goes through a validation step, where each task's constraint fields are analyzed for acceptability and validity, followed by task addition to the main schedule. On every processing cycle (every second by default), every task in the main schedule is sorted by each task's priority value, and then from the top, first each running task is checked to see if it is finished running or if it's running longer than expected, and afterwards, each pending task is checked to see if its constraints have been satisfied to be run, or if it has expired and needs to be removed. Figure 3 shows a flowchart of the operation of the SM.

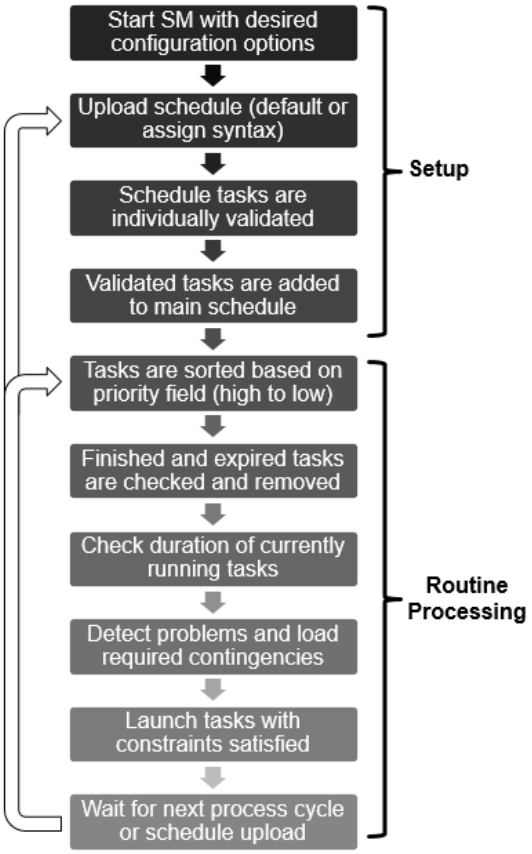

Figure 3: Schedule Manager Operation Flow

To facilitate working with and testing the SM, certain features were added mainly for testing convenience. For example, when the SM is started in standalone mode, configuration parameters can be passed to it on the command-line to control aspects such as constraint check overrides and configurations like the path to a system temperature file. After the SM is started and configured, it will either load a default schedule or wait until it detects a new schedule based on the file modification date. The SM accepts two main formats for schedules, the default being comma-separated task fields with one line per task and an alternative format where only desired fields need to be provided and are specified with assignment syntax (e.g., `id=1`, `priority=3`, etc.). With the schedule filetype being a comma-separated values (CSV) file, the default syntax is manageable by

editing the schedule in a spreadsheet such as Microsoft Excel, otherwise the assignment syntax is easier to manage when using a text editor. When the SM loads a schedule, it will perform task validation for each task in the schedule, making sure there are no syntax errors or invalid field values. Aside from the `status` and `pid` fields which are populated at runtime, Table 1 describes the task fields that can be provided for each task in the schedule.

Table 1: Schedule Manager Task Fields

Name	Description
status	The current status or return value of a function task
pid	The process ID returned when forking and executing a task with type 0
id	The identification number of the task, also includes task sequence ID
start_time	The earliest time that the task can execute in seconds since Jan 1, 1970
expire_time	The deadline for the task to execute in seconds since Jan 1, 1970
interval	The delay in seconds to add when rescheduling a completed routine task
duration	The expected duration of the task in seconds
conflict	The conflict categories the task belongs to, each bit corresponds to a category
priority	The priority of the task, higher priority tasks execute first
type	The task type (0: process, 1: function, 2: cFE message, 3: cFE function)
constraints	The index in the variable constraints table to associate with the task
args	The custom arguments that correspond to the task
mem_lower	The minimum free system memory in MB required for the task to execute
mem_upper	The threshold for free system memory in MB for triggering a contingency
store_lower	The minimum free system storage in MB required for the task to execute
store_upper	The threshold for free system storage in MB for triggering a contingency
power_lower	The minimum free system power in W required for the task to execute
power_upper	The threshold for free system power in W for triggering a contingency
temp_lower	The threshold for system temperature in °C for triggering a contingency
temp_upper	The maximum allowable system temperature in °C for a task to execute

Excluding the `args` task field, which can store a string argument, all other task fields are 32-bit integer fields. The `status` and `pid` fields are used when running a process or function task, respectively. The SM can keep track of running functions with the `status` field or monitor signals from a forked process with the pid stored in the `pid` field. The following task fields `id`, `start_time`, `expire_time`, `interval`, `duration`, `conflict`, `priority`, and `type` represent the core constraints of the SM. Following the core constraints, the `constraints` field links the task to an index in a separate variable constraints file which contains a list of task profiles and corresponding variable constraints (by default including memory, storage, power, and temperature constraints).

The `id` field specifies both an individual task's ID and the sequence it belongs to. By default, 0-999 is reserved for the individual task's ID, and the remaining digits represent the sequence ID (e.g. task IDs 5001, 5002, 5003 represent task 1, 2, 3 as part of sequence 5). Tasks that belong to a sequence require the previous task to complete successfully before running, and are automatically removed/rescheduled along with the other tasks in the sequence in cases such as task expiration/failure or routine task sequences.

The `start_time` field specifies the earliest system time that the task is allowed to start. The `expire_time` field specifies the latest system time the task is allowed to start, and triggers task removal/contingency if the task is unable to start by that time. The `interval` field specifies if and how often a task should be rescheduled after execution, with the value of the field corresponding to the delay between the task's completion and next required start time. The `duration` field specifies the longest that the task is allowed or expected to run, and is used to trigger warnings or contingencies in the event of unexpected delays.

The `conflict` field specifies which categories of resources or requirements the task requires. Every bit in the 32-bit `conflict` field represents a mutex and can represent many task requirements such as the availability of a sensor/actuator or modes of operation to block specific categories of tasks from running. Before a task is launched, its `conflict` field is compared with all other running tasks, and if any set bits overlap, the task is not allowed to run. The `priority` field specifies which tasks will attempt to start first. On every processing cycle, the SM sorts all tasks based on the `priority` field (with first-come-first-serve for tasks with the same priority), and therefore tasks with a higher priority will be able to start and obtain shared system re-

sources ahead of lower priority tasks which will need to wait their turn. The `type` field specifies the type of the task, and for the SM in standalone mode can represent either a process call such as an executable or script with value 0 or a function call with value 1. The most commonly used function call is the `clear_task` function, which can accept parameters to control which tasks are removed taking into account their run status, and also a result/status value in the case where another task or node is clearing the task. Other example function tasks include loading different types of schedules through a specified file path and killing the main SM process (useful for testing purposes). Additional options available for the `type` field when running in CSM will be described in the next section.

The `constraints` field specifies the variable constraints that are associated with the task. By default, a separate variable constraints file includes lower and upper bounds for memory, storage, power, and temperature values for a category of task to execute. This simplifies the main schedule syntax, enables tasks to share a variable constraints profile with other tasks, and allows for variable constraints to change in value at runtime (e.g., to change in levels of safety). For the example variable constraints, the lower bound of the memory, storage, and power constraints represent a requirement for task execution, while the upper bound of these constraints will generally be used for contingency tasks, such as if the value of one of these resources drops below a user-defined threshold. For the temperature constraint, the upper bound is instead used as a requirement for task execution (i.e., temperature must be below a certain value), and the lower bound is used for contingency tasks (i.e., temperature above a certain value triggers contingency tasks). Similar to the variable constraints file, a separate contingency file can also be used to enable contingency schedules to automatically be loaded when a task fails (i.e., returns a nonzero value or exit code), with each line in the file corresponding to a contingency schedule filename and the index of the loaded schedule corresponding to the task return value.

By properly assigning these described constraints to individual tasks in a schedule, a mission operator can have finely nuanced control over how the spacecraft will operate and respond to unexpected situations. The capability of assigning constraints to each individual task also enables multiple mission operators and schedule designers to share the system assuming critical task priorities and resources are discussed and agreed upon in advance. With a proper schedule design, a wide range of potential mission situations can be supported and can result in both optimized operations (e.g., reducing the need for arbitrary scheduled buffer times between conflicting tasks) and safer operations (e.g., preventing destructive task conflicts from occurring due to unexpected delays or schedule design oversights). The next section will describe how the SM can be integrated and used with cFS, in the form of the CSM app.

cFS Schedule Manager Overview

For missions designed with cFS, necessary functionality is developed as separate cFS apps that communicate using the cFE SB. Similarly to how SCH, SC, and LC trigger mission operations by sending SB command messages, the CSM needs to do the same to be compatible with the existing paradigm. To achieve this compatibility, two additional task types are added: the ability to send raw cFS messages (`type` 2) and to trigger cFS functions (type 3).

For typical cFS usage, the SC app enables a user to define SB command messages in tables with absolute or relative time offsets for execution. After compiling the table into an SC-compatible format, either a Relative Time Sequence (RTS) or Absolute Time Sequence (ATS) schedule, or optionally creating a schedule text file compatible with the relatively newer Stored Commands Absolute (SCA) app, schedules can be loaded on the fly using the appropriate SB command message. The CSM can represent SB command messages in its schedule similarly to the SCA app, with the hex of the command represented in ASCII in the `args` field. If the task is assigned `type` 2 and has a properly formatted SB command, the message will be put on the SB at the specified time and following all the constraints associated with the task as usual. This method of sending SB command messages following specified constraints can be used alongside SC and/or SCA and trigger the start command for their respective schedules with a higher degree of controllability.

For more nuanced control, such as SB command messages that should be formatted on the fly given the state of the spacecraft, functions can be built within the CSM that are called when the task is assigned `type` 3. These functions can be used to design any capabilities not feasible or simple with predefined cFS commands. Building reusable mission operations within CSM functions directly can also simplify the required schedule design for the mission, although at the cost of flexibility of the involved operations.

To achieve the effect of monitoring and control-

ling cFS tasks with duration, some form of compatibility needs to be designed with the system, whether it is on the CSM side or the controlled cFS app side. For cFS apps with predictable telemetry outputs, a CSM function can be built that subscribes to telemetry from the controlled cFS app to detect the `status` and return value of the task. Alternatively, a simpler method that does not require custom CSM functionality is for the controlled cFS app to directly alert CSM when its task is finished and its `status` through the scheduling of a `clear_task` function command.

Multi-Spacecraft Coordination through Availability Prediction

To predict resource availability and detect potential issues with the main schedule, an availability simulation function can be called with a `conflict` value as an input parameter. When called, this function creates a copy of the main schedule and simulates a period of time (10 minutes by default), keeping track of the combined `conflict` task field of all running tasks. Tasks are assumed to be completed a set amount of time after they are started based on their duration field. Every second, the input `conflict` field parameter is compared with the combined `conflict` value of all running tasks, and if there is no overlap, availability is set to 1; otherwise, it is set to 0. For the default availability array of size 20, each `int` in the array stores 30 seconds of availability info by setting the 30 least significant bits according to the calculated availability values. If any tasks expire without being able to run during the simulation, a warning is issued which can potentially be acted upon with a custom contingency plan if desired. Even if the calculated availability array is not needed, the availability simulation function is still beneficial to run routinely to detect potentially expired tasks ahead of time.

Using the capability to calculate availability windows for specific `conflict` field values, a spacecraft can predict the availability of its peer spacecraft by sending requests for availability windows. For example, if it is desired to schedule a synchronized data collection event, a spacecraft can send an availability request to all other spacecraft in the cluster providing a `conflict` value representing the resource requirement for the data capture task (e.g., relevant sensors available and spacecraft not busy maneuvering) and then, based on the earliest time acceptable for all spacecraft, schedule the task among all spacecraft. Figure 4 shows an example of how a task with varying duration values can be scheduled given the availability windows of four spacecraft.

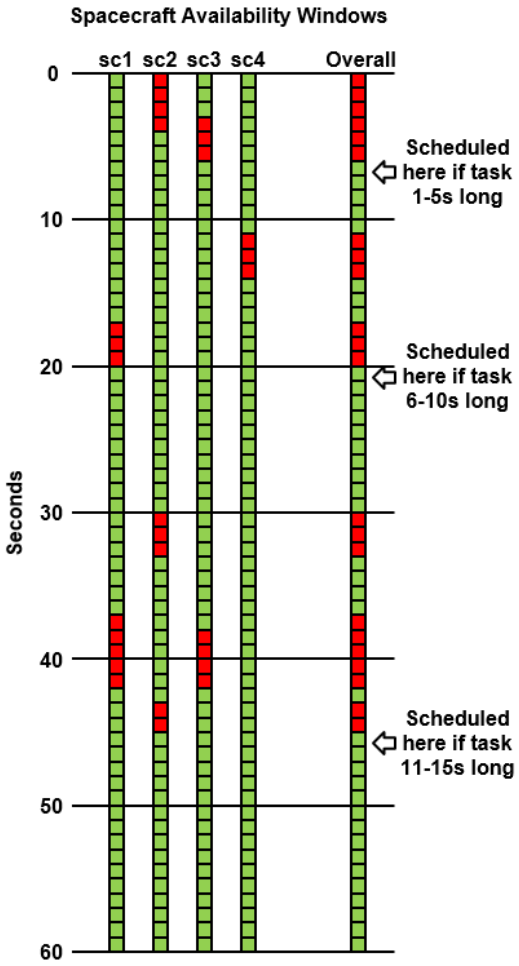

Figure 4: Coordinated Task Scheduling with Spacecraft Availability Windows

To accomplish a synchronized scheduled event, a networking infrastructure needs to be established. Using the CSM and a custom-built cFS communication app that can send SB command messages to specified IP addresses, a function that creates a message specifying source IP, destination IP, timestamp, `conflict` value, and availability request can be used to trigger a CSM function on a destination spacecraft, which will then run the availability simulation function and return a similar message with the availability window to the source IP. Using this infrastructure, spacecraft in a constellation can schedule tasks amongst each other and improve their overall synchronization and collaboration capabilities. An example of this capability is described in the demonstration section.

TESTING AND CODE ANALYSIS

For spacecraft flight software, adequate code testing is a formal requirement for many high-profile and safety-critical missions. There are many methodologies that are used for code testing, but the overall goal is to show that all parts of the code are tested and that there are no flawed edge cases. For the cFS apps released by NASA GSFC on GitHub, unit tests are provided leveraging a NASA-developed unit testing framework called `UT-Assert`. A newer proof-of-concept coverage test was recently released in their cFS sample application repository which provides a method for running cFS app unit testing and outputting code coverage results. Following the example provided by legacy cFS tests along with the newer testing implementation, it is currently possible to build cFS app tests that will likely conform to the new standard following the release of the next major version release of cFS, cFS Caelum (v7.0).

However, due to the design of the SM and its flexibility for schedule execution, unit testing alone is not sufficient to be confident of proper functionality. To completely exercise the SM, end-to-end functional tests are required, including comparisons to prior established correct outputs to ensure no mechanisms have broken. Compared to unit tests which ensure each function operates exactly as designed, functional tests ensure the whole system functions together in a predictable manner given a range of acceptable inputs. By combining these two techniques as well as including code analysis tools such as `gcov` and `lcov` to detect the code coverage percent achieved through tests, a higher degree of confidence can be gained as to the reliability of the built code.

Unit Testing

The simplest application of unit tests is to provide arguments to a function and ensure the output matches what is expected. Unit testing frameworks such as `GoogleTest` add convenience tools to run functions and check their outputs, potentially with mock functions to assist with setting up tests. For unit testing the SM standalone, due to the SM library functions not following a typical method of input/output and often reading and writing to structure and array pointers instead of returning values, unit tests in a traditional sense would be impractical and inefficient. For example, one method to implement unit tests would be to call the SM function to load a schedule, and then run tests for each field in the loaded schedule (i.e., every field in the array of task structs). This procedure would result in a large amount of code and would not add anything that a functional test with proper coverage would not offer. The next section will cover how the SM uses functional testing combined with coverage analysis to accomplish the same goal in a more efficient manner.

For unit testing the CSM, the example coverage test provided by the cFS sample app can be used as a template to validate the cFS app structure of the CSM. Because the CSM does not include any major differences to the standard template and simply adds custom functions and command code handlers, adapting the coverage test is not challenging and adds some assurance that there are no issues introduced to the cFS-specific portions of the CSM app. The updated version of the cFS framework also includes the ability to run all unit tests and generate coverage reports with `lcov` directly using the main makefile.

Functional Testing

To further ensure proper functionality for the SM, accelerated functional testing can be used to ensure outputs match expected values for a collection of schedules developed and tested previously. Using a top-level functional test script, the SM can be started with predefined schedules loaded to capture the log output. By comparing the log outputs to logs previously manually validated for correctness and saved, schedule correctness can be verified after making modifications to tangential SM functionality without requiring repeat manual scrutiny.

To ensure schedule outputs are the same and to accelerate functional testing, the Linux `faketime` utility can be used to start the SM instance at a simulated time as well as run at a specified rate. Through testing, a speedup value of `x40` sometimes resulted in inconsistent results (with task start/end times occasionally being off by a second), but at `x30` the results were sufficiently consistent, resulting in significant time savings for running through the list of built functional tests. Figure 5 below shows an example functional test and execution output to validate the functionality of the `start_time` task field.

By adding the `fprofile-arcs` and `ftest-coverage` flags when compiling the SM with GCC, the number of times each line in the code is run can be tracked. `gcov` can be used to generate coverage data and along with `lcov` and `genhtml` to generate a human-readable coverage report showing any lines in the code that were never run. Figure 6 shows the coverage report generated after using the functional-test script to run through all functional

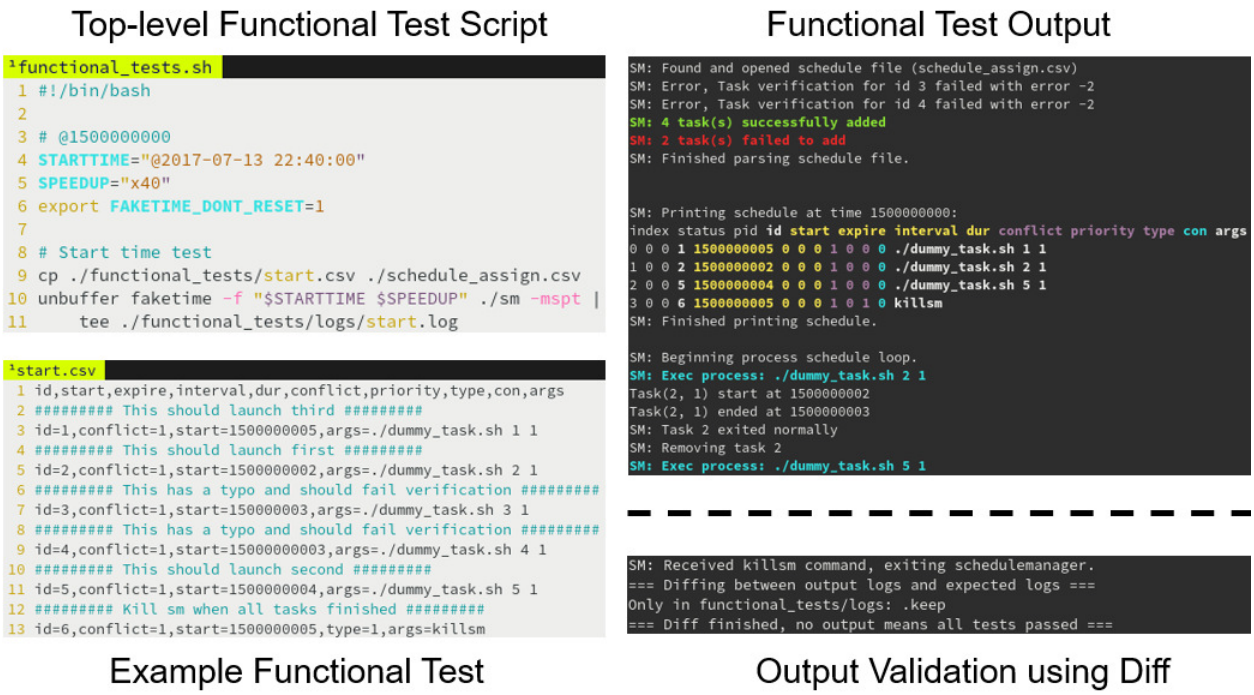

Figure 5: Functional Test and Example Schedule (Left), and Execution Output (Right)

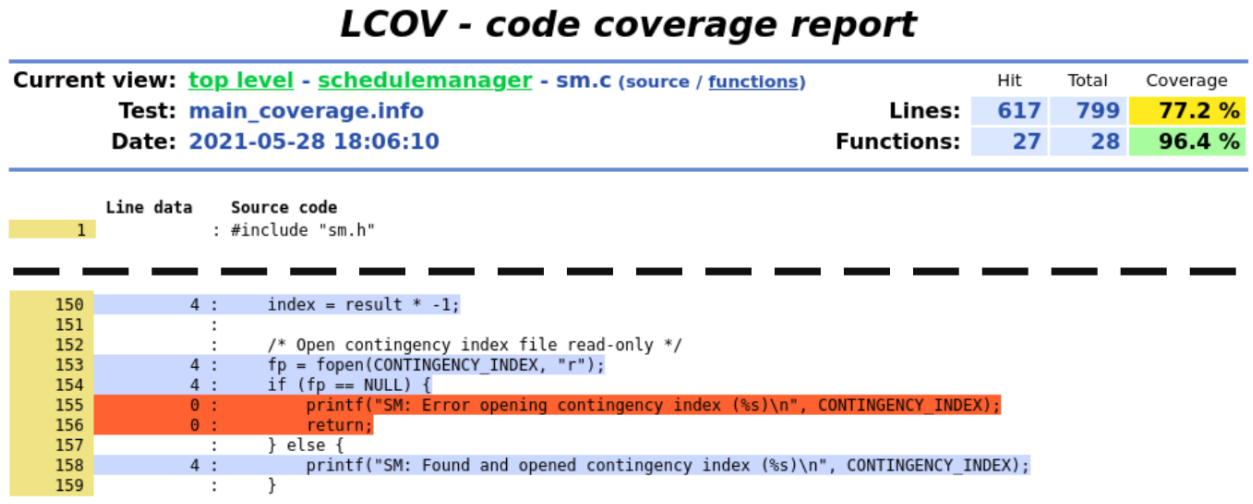

Figure 6: Coverage Report Snippet for Schedule Manager After Functional Tests

tests.

By scrolling through the report, one can immediately identify which lines of code have not been tested. In Figure 6, coverage percent values for both lines and functions are shown in the top right, and functional code lines that have not been run are highlighted in red. In this example, the error condition for the contingency index file being unable to be opened is never tested. Because the SM is designed to check a hard-coded path for this file, an easy method of testing this line would be to rename/remove the contingency index file directly in the top-level functional test script and have it replaced before the next test. By creating and adding new schedules to the functional test runner and using a script to recompile the SM, rerun the functional tests, and regenerate the coverage report, it is possible to rapidly and iteratively improve code test coverage compared to the unit-test method.

To validate CSM functionality with functional tests, the process is more complex due to having to validate the command and telemetry interfaces to the CSM app. To assist with sending commands and receiving telemetry, the COSMOS ground station can be used. COSMOS includes script testing capabilities which allows the user to automate the sending of commands and parsing of resulting telemetry, with any errors automatically detected and printed. COSMOS also provides a command sequence tool that can be modified to generate cFS table schedules for the SC app (a potential target for CSM tasks). The demonstration section below will show some examples of functional tests through actual mission scenarios.

DEMONSTRATIONS

To show the practical capabilities of the SM and CSM, demonstrations were developed for various platforms. In the first demonstration, the capability of CSM for managing the operations on the CASPR mission will be described. The second demonstration will demonstrate cluster data distribution on VMs. The final demonstration will demonstrate constellation coordination in 42 using simulated availability windows with 42 and on a PYNQ-Z2 cluster.

STP-H7-CASPR Experiment

Although CASPR was developed with cFS, only a few of the onboard experiments were designed to be operated through cFS apps due to the redundant safeguards put in place in the case of system failure, the risk-tolerant nature of the mission, and the convenience and flexibility needed by users to simply build shell scripts to control their experiments.

With CASPR being mounted onto the ISS, even if a process external to cFS crashes and the system goes in an indeterminate state, the mission is not at risk because there are no mechanisms like propulsion or battery requirements that could result in catastrophic failure. CASPR also leverages multiple hardware and software resilience technologies such as watchdog management and multi-boot. Using the watchdog management capability provided by cFE's PSP and a radiation-hardened hardware watchdog on the CSP, the head node of CASPR can be rebooted if cFS ever becomes unresponsive and loses the ability to signal heartbeats to the watchdog. Additionally, by using the multi-boot capability of the Zynq SoC along with redundant boot images that are signed using an RSA key stored in the eFUSE register of the Zynq, any image corrupted due to radiation will be skipped, further improving the reliability of the system. After analyzing the tradeoffs relating to software capability, convenience, and stability, it was decided by the CASPR software team that it was an acceptable risk to allow experiments to be controlled via scripts external to cFS (which would typically not be allowed for traditional missions).

Therefore, the software design of CASPR uses cFS to manage communications with the STP pallet and manage critical services such as telemetry and file uplink/downlink, while other experiment capabilities are controlled through shell scripts. CASPR includes many different payloads including a compact binocular telescope to capture high-resolution and low-GRD (ground-resolved distance) NIR(Near-Infrared)/RGB images, a gimbal motor to position the telescope optics, a neuromorphic event-driven sensor, and an AMD GPU SoC. Along with multiple processor boards including a CSP, two SSPs, and a µCSP using a Microchip SmartFusion2 SoC, CASPR would exceed its power budget if all subsystems were powered on concurrently. It would also be beneficial to routinely run specific experiments at specified times and remove the risk of potential overlaps due to periodic experiment activation intervals.

By using the CSM with tasks specifically referring to one shell script per experiment, conflicting experiments can be determined in advance and, with proper constraint specification, the risk of any accidental, destructive overlap can be mitigated. For example, experiments can configure the imager and collect images without considering potential conflict with a routine image capture script. Also,

each of the variable constraints described in the SM overview are critical to manage on CASPR. For example, images are captured with 4K resolution, and the super-resolution experiment requires dozens of images, representing hundreds of megabytes of combined RAM and flash memory, to execute. Along with the power and temperature constraints to ensure that CASPR operates only under acceptable conditions, the CSM has significant potential to improve the capabilities and science return of the CASPR mission.

Virtual Machine Cluster Testbed

Towards the rapid prototyping of advanced distributed space mission concepts, many applicable software tools exist that can be used together to create a complete end-to-end test system completely in software. The Yocto Project enables the creation of custom, lightweight operating systems with community support for Xilinx Zynq-based platforms due in large part to its use in the PetaLinux development tool by Xilinx. One benefit to Yocto is the ability to have parallel builds for multiple platforms (e.g., x86 and ARM) with minor changes necessary to a local configuration file. After generating an x86 image with Yocto, a Virtual Machine Disk (VMDK) can be exported which can then be loaded by virtualization technologies such as Oracle VM VirtualBox. One benefit of using cloned virtual images is the relatively low complexity in making system-configuration changes to individual machines such as changing startup operations without requiring image regeneration typically necessary for other approaches like with Docker.

A configurable cluster of networked VMs based on a single VMDK can quickly be launched from the command-line by writing a script that uses the VirtualBox command-line utility `VBoxManage` to clone the VMDK a desired number of times, create individual VMs with specified resources and configurations such as a serial port connection, and then configure the network and startup files after boot. With proper configuration, the resource requirements necessary can be minimized, with required memory and storage being well under 256MB per VM. Using this method, within minutes a network virtual testbed can be set up from scratch with a configurable number of nodes and the latest software resources to be tested.

After starting cFS on each target, tests can be performed by using the command-line test capabilities of COSMOS. After defining command and telemetry definitions in COSMOS, individual commands can be referenced by name to be sent using a Ruby script file, and similarly, individual telemetry fields can be checked to validate successful operation. For example, a Ruby test file can be built that checks a telemetry command count field for a specific cFS app, sends one or multiple commands to the app, and then verifies that the command count field went up by the expected number. By being able to set up and test a virtual cluster running cFS from the command-line, software development overhead can be greatly reduced compared to graphical-based alternative workflows.

Using the approach described above, a virtual constellation testbed was built, featuring the Better Approach to Mobile Adhoc Networking (B.A.T.M.A.N) protocol to create a mesh network between the VMs as well as the UDP-based File Transfer Protocol (UFTP) daemon to enable lossy unidirectional multicast file transfer. The mesh network capability offered by B.A.T.M.A.N. enables the creation of a decentralized network with dynamic routing table changes, perfectly suitable for a constellation of satellites that may change in position, resulting in the availability of network routes fluctuating constantly. UFTP enables file transfer over a lossy network (e.g., radio communications over long distances) by splitting the file transfer into multiple phases, with an initial transfer attempted with UDP, and recipients requesting specific corrupted or missing packets until the transfer is successful. Combined, this approach enabled the rapid and replicable creation of a cluster of VMs in a realistic, flight-like configuration with cFS started on boot and with COSMOS to validate networked communication and file transfer over a mesh network.

Constellation Coordination in 42 Simulator on PYNQ-Z2 Cluster

For a satellite constellation, coordinated actions can be useful for various reasons. In general, synchronized sensor readings is a commonly useful and often required capability mentioned for constellation missions. To demonstrate this capability with the previously described availability simulation approach, a cluster of simulated spacecraft running cFS and CSM were realized on PYNQ-Z2 development boards (based on the Zynq-7020 and similar to the CSP) and represented in the 42 simulator.

For the demo including four spacecraft, one spacecraft was designated as a leader node, representing a node that might have a specialized sensor or some method of determining the ideal timing to capture a synchronized sensor reading. Using

a custom cFS app to enable socket-based node-to-node communication, CSM on the leader node was able to use the procedure described in the Availability Prediction subsection above to decide on a time and schedule an image-capture event in 42. To observe the effect of the command, the spacecraft in the constellation represented in 42 were issued rotation commands to point towards Earth (using instantaneous reorientation mode for convenience). Through this proof-of-concept, the feasibility of scheduling other synchronized actions can be seen. Figure 7 shows a picture of the satellite constellation in 42 along with a view from one of the spacecraft.

Figure 7: Constellation of Four Satellites in 42 Simulator (Top), Spacecraft View from Lead Satellite (Bottom)

DISCUSSION

Due to the complexity associated with the practical development of autonomous mission management solutions, especially solutions applicable to multi-vehicle missions, it is difficult to design an autonomous mission manager that is general-purpose. Even the design of a solution to a specific targeted constellation mission can grow in complexity to an infeasible level without a proper, modular solution to accommodate potential situational conflicts. Without explicitly checking an individual task's constraints before running the task, the alternative is to ensure excess resources are available on the system and separate potentially conflicting tasks with large buffers of time in between. Not only is this solution not fool-proof, it also is much less efficient and adds complexity to the schedule validation and approval process.

The downside to assigning constraints to each individual task is that it may be excessive in certain situations, especially in simple scenarios, and may be an inefficient solution for conflict resolution in more complex scenarios. In the case of simple scenarios, the flexibility in schedule syntax for the SM (either the default comma-separated field values or the field-assignment syntax) and the consistency in field types (`int32`) and default values (0) aims to address lowering the complexity when it is not necessary. In the case of complex scenarios, the goal of the SM and CSM is to provide a potentially feasible baseline solution which can then be iterated on as needed. The schedule design process will likely be the main bottleneck when designing more complex missions, and the construction of an automated planning and schedule-generation tool would aid in this process.

As future work, improvements to both the schedule design and testing framework are planned. An early iteration of this research included a Python Tkinter GUI to assist with schedule creation by including a scrollable menu of template tasks that could be selected to populate field values to be optionally modified with editable dropdown fields for every constraint before appending to a specified schedule file. However, this approach ended up being more of a maintenance burden than a benefit, with a simple copy-paste from existing schedule tasks often resulting in a more rapid schedule-creation method. Unfortunately, relying on the manual approach of generating schedules increases the learning curve difficulty for schedule creation for those less familiar with the project and loses the potential benefits of automatic task suggestions, syntax checking, and validation that could be added in a tool. For the testing framework, improvements will focus on outlining specific requirements and linking them to specific functional tests, similarly to the reports generated for various NASA GSFC cFS app tests. Additionally, the coverage percent achieved should be at or close to 100%, which would add an additional layer of assurance that the software is safe for flight.

CONCLUSION

Through the above-described tests and demonstrations, the capabilities and functional mechanisms of the SM and CSM are overviewed. The goal of this research is to contribute to the development and improvement of the next generation of autonomous space missions. The concepts described in the design of the SM aim to establish a general-purpose baseline that is a step-up from traditional mission management without any level of autonomy. In exchange for the capabilities potentially offered by more sophisticated autonomy solutions leveraging technologies such as heuristics or neural networks, the relatively simplistic solution achieved by the SM offers a comprehensible general-purpose solution that, at minimum, can act as a baseline for comparison to justify the need for more sophistication.

The design of the CSM represents a demonstration of how the SM concept can be integrated into other flight-software frameworks. The benefits offered by leveraging an open-source flight-software framework are immense, and can be indispensable if the goal is to learn and benefit from others in the space community. By improving the autonomous capabilities of an open-source flight-software framework like cFS, the probability is much higher for the effort to lead to the successful deployment of a mission such as a collaborative constellation of small satellites.

Acknowledgments

This research was funded by industry and government members of the NSF SHREC Center, the National Science Foundation (NSF) and its IUCRC Program under Grant No. CNS-1738783, and NASA STTR contracts NNX16CG21P and 80NSSC18C0178. The authors would like to thank Brendan O'Connor from Emergent Space Technologies for providing thorough guidance regarding the design of the SM, as well as everyone else at Emergent who provided feedback, Christopher Wilson from NASA GSFC for initial project guidance and support, and Rachel Misbin from SHREC for developing the 42 modifications and contributing to the demonstrations.

References

[1] Jacqueline Le Moigne, John Carl Adams, and Sreeja Nag. A new taxonomy for distributed spacecraft missions. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13:872–883, 2020.

[2] Carles Araguz, Elisenda Bou-Balust, and Eduard Alarcón. Applying autonomy to distributed satellite systems: Trends, challenges, and future prospects. *Systems Engineering*, 21(5):401–416, 2018.

[3] Christopher Wilson, Jacob Stewart, Patrick Gauvin, James MacKinnon, James Coole, Jonathan Urriste, Alan George, Gary Crum, Elizabeth Timmons, and Jaclyn Beck. Csp hybrid space computing for stp-h5/isem on iss. In *29th Annual AIAA/USU Conference on Small Satellites*, 2015.

[4] Sebastian Sabogal, Patrick Gauvin, Brad Shea, Daniel Sabogal, Antony Gillette, Christopher Wilson, Alan George, Gary Crum, Ansel Barchowsky, and Tom Flatley. Ssivp: Spacecraft supercomputing experiment for stp-h6. In *31st Annual AIAA/USU Conference on Small Satellites*, 2017.

[5] Seth Roffe, Theodore Schwarz, Thomas Cook, Noah Perryman, Justin Goodwill, Evan Gretok, Aidan Phillips, Mitchell Moran, Tyler Garrett, and Alan George. Caspr: Autonomous sensor processing experiment for stp-h7. In *34th Annual AIAA/USU Conference on Small Satellites*, 2020.

[6] David McComas, Jonathan Wilmot, and Alan Cudmore. The core flight system (cfs) community: Providing low cost solutions for small spacecraft. In *30th Annual AIAA/USU Conference on Small Satellites*, 2016.

[7] L. Kepko, Chuck Clagett, L. Santos, Behnam Azimi, D. Berry, T. Bonalsky, D. Chai, Matthew, Colvin, Alan Cudmore, A. Evans, Scott Hesh, S. Jones, J. Marshall, N. Paschalidis, Zach, Peterson, J. Rodriquez, M. Rodriquez, Salman Sheikh, S. Starin, and E. Zesta. Dellingr: Nasa goddard space flight center's first 6u spacecraft. In *31st Annual AIAA/USU Conference on Small Satellites*, 2017.

[8] Larry Kepko, Luis Santos Soto, Chuck Clagett, Behnam Azimi, Dean Chai, Alan Cudmore, James Marshall, and John Lucas. Dellingr: Reliability lessons learned from on-orbit. In *32nd Annual AIAA/USU Conference on Small Satellites*, 2018.

[9] H. Cannon, P. Berg, A. Bajwa, and A. Crocker. Ladee preparations for contingency operations for the lunar orbit insertion maneuver. In *2015 IEEE Aerospace Conference*, 2015.

[10] Danilo José Franzim Miranda, Maurício Ferreira, Fabricio Kucinskis, and David McComas. A Comparative Survey on Flight Software Frameworks for New Space Nanosatellite Missions. *Journal of Aerospace Technology and Management*, 11, 00 2019.

[11] David McComas and Ryan Melton. Opensatkit enables quick startup for cubesat missions. In *31st Annual AIAA/USU Conference on Small Satellites*, 2017.

[12] Steve Chien, Rob Sherwood, Daniel Tran, Benjamin Cichy, Gregg Rabideau, Rebecca Castano, Ashley Davis, Dan Mandl, Stuart Frye, Bruce Trout, Seth Shulman, and Darrell Boyer. Using autonomy flight software to improve science return on earth observing one. *Journal of Aerospace Computing, Information, and Communication*, 2(4):196–216, apr 2005.

[13] Steve Chien, Joshua Doubleday, David R. Thompson, Kiri L. Wagstaff, John Bellardo, Craig Francis, Eric Baumgarten, Austin Williams, Edmund Yee, Eric Stanton, and Jordi Piug-Suari. Onboard autonomy on the intelligent payload experiment cubesat mission. *Journal of Aerospace Information Systems*, 14(6):307–315, 2017.

[14] 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*, 15(3):107–119, 2018.