Spacecraft Mission Agent for Autonomous Robust Task Execution

by

Antony Gillette

B.S., University of Florida, 2016

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Antony Gillette

It was defended on

April 1, 2019

and approved by

Alan George, Ph.D., FIEEE, Department Chair, Professor
Department of Electrical and Computer Engineering

Zhi-Hong Mao, Ph.D., Professor
Department of Electrical and Computer Engineering

Jingtong Hu, Ph.D., Assistant Professor
Department of Electrical and Computer Engineering

Thesis Advisor: Alan George, Ph.D., FIEEE
Department of Electrical and Computer Engineering

Spacecraft Mission Agent for Autonomous Robust Task Execution

Antony Gillette, M.S.

University of Pittsburgh, 2019

Autonomy in space systems can drastically reduce the workload of ground crews for satellite missions, especially for clusters of satellites. Additionally, autonomy can increase the efficiency of missions by maximizing the utilization of resources and rapidly handling any issues that arise without having to wait for instructions from the ground. This research presents an agent-based, task-execution approach to onboard spacecraft autonomy. Instead of the traditional approach requiring onboard planning and scheduling, this method uses a combination of constraint and priority parameters associated with every task to ensure robust task execution with behavior as intended. Using this method, tasks will only run under safe conditions (e.g. no conflict with any running tasks), which enables conflicting tasks to be scheduled closer together or even overlapping for lower-priority tasks. This approach manages the execution of tasks on the timescale of seconds, enabling conflicting tasks to run sequentially, thereby increasing productivity if earlier tasks finish ahead of schedule. This framework leverages the NASA-developed, open-source projects cFE and PLEXIL and was tested on development boards comparable to flight hardware.

Table of Contents

List of Tables

List of Figures

Preface

I would like to thank Dr. George for his guidance and direction throughout this research as well as my committee members. I would also like to thank the following people for discussion, feedback, and suggestions regarding my research: Brendan O'Connor and John McGreevy from Emergent Space Technologies, Christopher Wilson and Gary Crum from NASA Goddard, and Jeremy Frank from NASA Ames. Thanks to Christopher Manderino for building the Hacksat for the demo, Tyler Garrett for helping with Thesis formatting, and Thomas Cook and Sebastian Sabogal for feedback regarding potential mission scenarios and power constraints for CASPR. 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 contract NNX16CG21P.

This thesis is extended from a conference paper published in March 2018 [1].

1.0 Introduction

There is a growing interest in small spacecraft by commercial and government organizations to meet critical observations and measurement requirements. These organizations desire to use small spacecraft in more sophisticated configurations, such as constellations and coordinated measurements [2]. Most solitary spacecraft are operated by a ground crew; however, it is challenging to maintain and manage operations for clusters of satellites without proportionally increasing the number of required ground operators. In space systems, increasing the level of autonomy for a spacecraft, or cluster of spacecraft, can drastically reduce the workload of ground crews required for satellite missions. An additional benefit of autonomy is an increase in the efficiency of missions by maximizing the utilization of resources and handling any issues that arise without having to wait for instructions from the ground.

This thesis presents a simple, yet powerful spacecraft mission agent, the Spacecraft Mission Agent for Autonomous Robust Task Execution (SMAARTE), for autonomously managing the execution of tasks on one or more spacecraft. Traditionally, autonomy in space systems consists of an onboard planner used to determine the optimal path to achieve mission goals and an onboard scheduler used to add tasks to the spacecraft schedule table according to the planner's instructions. Using a planner and scheduler in this manner results in a schedule that requires replanning and rescheduling if any unanticipated events occur, such as a system malfunction or a task running longer than expected. SMAARTE uses a combination of constraints and priorities to create an agent that can efficiently manage the execution of tasks and automatically handle issues that arise without necessarily requiring replanning or rescheduling.

By associating constraint and priority parameters with every scheduled task, the agent can handle unexpected situations as they arise, using the same procedures as a ground operator. Time constraints restrict tasks to run during specified time-windows, enabling start-time flexibility in the case of delays while also preventing tasks from running when they are not permitted. Conflict constraints keep tasks from running if they interfere with another task (e.g. resource contention or task dependency). Utilization of these constraints can minimize the delay required between sequential tasks, thereby increasing the potential number of task executions feasible in a given time-window. Priority parameters enable tasks to be prioritized based upon sensor results, such as detection of significant science data, or if any anomalies are detected. Priority parameters can also be used to override originally planned tasks with higher-priority tasks (e.g. error handling or maneuvering) and for determining when it is acceptable to end any conflicting tasks early. This usage of constraints and priorities simplifies the construction of an autonomous system that will behave as desired, and it also facilitates the addition and modification of tasks from the ground.

The core of the SMAARTE framework, the Schedule Manager, was developed as a library, enabling it to be easily integrated into existing executives or simply run stand-alone. SMAARTE was developed alongside and integrated with NASA Goddard's core Flight Executive (cFE) and NASA Ames' Plan Execution Interchange Language (PLEXIL) executive to demonstrate a case study on hazard detection using a cluster of satellites and synchronized, camera-event scheduling. In this case study, a forest-fire event was simulated and the testbed cluster representing flight hardware performed a synchronized reaction that was triggered and coordinated. SMAARTE was also used to demonstrate conflict, priority, and routine execution functionality for various other scenarios, including simulated and real mission scenarios.

2.0 Background

The main end goal for this research is to enable autonomous capabilities for distributed space missions (DSMs). The SMAARTE framework was developed to be a component for the Distributed Automation Suite for Heuristic Execution and Response (DASHER) project, which is led by Emergent Space Technologies in collaboration with the National Science Foundation (NSF) Center for Space, High-performance, and Resilient Computing (SHREC) at the University of Pittsburgh. DASHER seeks to improve the coordination between the executive agents on a cluster of satellites while enabling autonomous operation.

2.1 Core Flight Executive (cFE)

To accomplish the goals established by DASHER, the first design decision made was to use cFE as a base for the project due to its flight heritage on several missions [3] and wide community acceptance. One of the most attractive features of cFE is the availability of an open-source release, which makes it one of the few options available for a standardized flight executive. cFE provides basic mission control functionality such as a command ingest, telemetry output, and a software bus for cFE application communication. The command and telemetry applications provide boilerplate mission management functionality, and the software bus enables the user to add more applications to interface with the system using a publish-subscribe mechanism. An additional benefit of cFE is its networking capabilities, which include several options such as the

Software Bus Network (SBN) and more recently the Software Bus Distributed (SBD) [4]. cFE also works on top of the OSAL (Operating System Abstraction Layer), which enables cFE to work on Real-Time Operating Systems (RTOS) for missions requiring low jitter and determinism.

2.2 Planning-Language Selection

After selecting cFE for inclusion in the framework, the next step was to identify if there was a pre-existing, community-wide, and decisive planning-language that could be used to facilitate the design of autonomous functionality. After an extensive literature survey, it was determined that no single planning-language was foremost dominant to flight systems. Of the planning-languages identified, a decision, analysis, and resolution chart comparing desirable features of a language (e.g. licensing, build difficulty, etc.) was made during the initial selection phase, shown in Figure 1.

	Licensing	Activity	Syntax Difficulty	Build Difficulty	Previous Projects	Flexa - bility	Has GUI	Example Code
PLEXIL	BSD	High	Med	Low	Mostly Related	Med	Yes	High
SPELL	GPLv3	Med	Low	High	Related	Med	Yes	Low
ASPEN /CASPER	N/A	Med	N/A	N/A	Related	Med	Yes	N/A
SCL	Proprietary	Low	Low	N/A	Related	High	Buildable	N/A
EUROPA	NOSA	Med	Low	Med	Moderately Related	Med	Yes	High
Python	PSFL	High	Low	Low	No baseline	High	Buildable	High
NMP MM	CHREC	Low	N/A	N/A	Moderately Related	Med	Buildable	N/A
Timeliner	Proprietary	Low	N/A	N/A	Mostly Unrelated	Med	Yes	N/A
PDDL	Varies	Med	High	N/A	Not Related	Med	Some	Med

Figure 1 Planning-Language Comparison Chart

After comparison with the key project goals, many of the options were eliminated due to critical development obstructions, such as licensing issues and integration difficulty. For this reason, these options were not obtained and tested, resulting in blanks in the chart. Since cFE is open-source, an accompanying open-source planning-language was naturally preferred to facilitate future code distribution. The requirement of needing to work on top of cFE also reduced the viability of options with complex stand-alone platforms like the Robot Operating System (ROS) framework. Also, as the target platform was the ARM processor architecture, the build complexity was a key factor. In the concluding analysis, the decision was narrowed down to PLEXIL and Python (using the Advanced Python Scheduler module), with PLEXIL eventually being selected due to the difficulty of formally verifying Python code and in consideration of the overlapping user base with the space community for PLEXIL.

2.3 Plan Execution Interchange Language (PLEXIL)

PLEXIL is a plan-execution framework with development led by NASA Ames. The source code is open-source and publicly accessible through SourceForge [5]. PLEXIL uses the concept of node trees to represent complex plans. The deterministic execution of a system can be controlled using a combination of different types of nodes, node states, and node transitions. A primary use case for PLEXIL is the autonomous operation of rovers such as the K10 rover. In the planning-language survey, PLEXIL won over the other choices due to its open-source license, simplicity to setup on the ARM processor architecture, determinism, and formal verification.

2.4 SHREC Processor Development, STP-H5, STP-H6, and STP-H7

Before SHREC was founded, the previous center name was the Center for High-Performance Reconfigurable Computing (CHREC). At CHREC, a high-performance hybrid space processor combining radiation-hardened and commercial-off-the-shelf components called the CHREC Space Processor (CSP) was developed in 2014 [6]. This eventually led to development of multiple missions by SHREC for the Department of Defense's Space Test Program starting with STP-H5 [7].

After the successful launch of STP-H5 on SpaceX CRS-10 in February 2017, the next experiment developed at CHREC was the Spacecraft Supercomputing for Image and Video Processing (SSIVP) experiment which was to target the STP-H6 mission [8]. It was during this time that the CHREC center finished its 11 years of operations and the SHREC center was formed. The Micro CSP (uCSP) was developed during the preparation for this mission. The SSIVP payload is currently delivered to NASA Kennedy and is scheduled to launch on SpaceX CRS-17 in April 2019.

The mission planned after this is the Configurable and Autonomous Sensor Processing Research (CASPR) experiment (not to be confused with JPL's CASPER below). The CASPR mission is currently under development at SHREC in preparation for STP-H7, including development for the center's next-generation SHREC Space Processor (SSP). This experiment will also likely include a high-resolution lens and camera, Neuromorphic camera, GPU accelerator, and gimbal for camera orientation.

The software architecture of STP-H5 makes it difficult to use SMAARTE for its mission operations due to being designed with a static cFE architecture, so mission control using SMAARTE will be only for the experiments SSIVP and CASPR. More details on SSIVP and CASPR mission scenarios will be described in the results section below.

3.0 Related Work

One of the goals in developing the SMAARTE framework was to provide an alternative approach to traditional spacecraft autonomy (using onboard planning and scheduling) by using a simple, agent-based framework with rules for priority and constraints. Two different approaches to autonomy using onboard planning and scheduling are presented in this section, with comparison to the SMAARTE framework in the discussion section below. This section ends with a general overview of recent papers related to the topic of autonomy and scheduling to discuss the various cutting-edge approaches to this problem.

3.1 CASPER and SCL on EO-1

One of the most heavily cited papers in the field of autonomous spacecraft software describes the considerations for autonomy on Earth Observing One (EO-1) [9]. EO-1, which flies in Low Earth Orbit (LEO), was developed to autonomously detect notable events using onboard image sensors and appropriately respond with the proper procedure. Aside from the software used to process images for detecting interesting phenomena, onboard software was also used for replanning and execution. EO-1 used the Continuous Activity Scheduling Planning Execution and Replanning (CASPER) software to handle planning (on the order of tens of minutes), and replanning when necessary by using feedback from the image-processing software. The output

from CASPER would then feed into the Spacecraft Command Language (SCL) executive, where the appropriate low-level commands would be robustly executed according to CASPER's provided plan.

Due to limited CPU resources, it was necessary for CASPER to vary the resolution of plans depending upon the proximity of events to the current time. For activities more than a day in the future, the plan would be abstract and not well-defined. CASPER would then plan these activities at a more detailed level as they got closer to their intended run time. The amount of activities EO-1 needed to handle per week was approximately 7800, which covered around 100 science observations. This number of activities resulted in detailed planning being restricted to 6 hours in the future to keep heap-space usage down.

3.2 Autonomous Mission Manager

Referencing EO-1's approach for autonomous capabilities, a more standardized approach to autonomy, using a modular autonomy architecture to improve on the reusability of autonomous capabilities (without being tied to specific hardware and software) is discussed in [10]. The research described is for the Autonomous Mission Manager (AMM) architecture sponsored by the Air Force Research Lab (AFRL), and it uses a Service-Oriented Architecture (SOA) to enable software to be partitioned into modules that can communicate using predefined data interfaces. Like EO-1, AMM uses CASPER for mission planning but replaces SCL with an executive provided by the Cooperative Intelligent Real-Time Control Architecture (CIRCA). AMM also adds a middleware inter-module messaging system called the Adaptive, Scalable, Portable Infrastructure for Responsive Engineering (ASPIRE) framework. The ASPIRE framework fills a

role similar to NASA Goddard's cFE by providing a messaging service between components, and it functions as an application wrapper that enables the software to not be restricted to a specific system or hardware. The goal of the AMM architecture is to standardize the data interface between the components mentioned above.

3.3 Recent Approaches to Autonomy and Scheduling

A general overview of approaches to autonomy for distributed satellite systems (DSS) is provided by Araguz et al. [11], along with trends, challenges, and future prospects. In this paper, a list of the issues and topics that autonomous satellite technologies target is discussed such as communication delays, reduced visibility with ground stations, improvement of science return, and mission robustness and tolerance against failure. These topics form the baseline of what current spacecraft autonomy researchers are attempting to solve. It is mentioned that autonomy is not an additional feature but rather the solution to operate dynamic and complex systems. The discussed approaches to Mission Planning Systems (MPS) are mainly all heuristic-based approaches like EO-1's approach which require replanning when detecting anomalies, with one exception being an "interesting approach" by Beauemet et al. [12] using a reactive algorithm that can produce instantaneous solutions based upon system rules. In this paper, Beauemet et al. use a similar approach to the approach used for the Schedule Manager described later in this thesis by assigning priorities, start time, duration, and resource requirements to every activity. However, the main difference is that they restricted the spacecraft to seven possible activities (such as data downlink, image processing, and reorientation) and ranked these static activities into a list of decreasing priority.

Without the restriction of requiring an autonomous mission manager to reschedule when anomalies are detected, many math-heavy scheduling solutions proving optimality can be found under the topic of Earth Observation Satellite (EOS) scheduling, with a good list of single-satellite approaches referenced in [13]. Similarly, a good list of references for multi-satellite optimal EOS scheduling can be found in [14]. These approaches are focused on proving optimality assuming no autonomy or replanning is required, which can be a valid solution for certain mission scenarios.

In the space field, there are few papers to reference that use the concept of scheduling tasks that are interdependent. However, in the general field of distributed computing, there are an abundance of papers that discuss scheduling for interdependent tasks with many related works cited in [15]. Similarly, without the restriction of space, there are many papers related to scheduling and autonomy for other platforms such as for Autonomous Underwater Vehicles (AUVs) like in [16] and for Unmanned Aerial Vehicles (UAVs) like in [17]. These systems do not have the same constraints as space systems though and often have a different focus and objective such as for path-planning and object-detection. These papers can however provide inspiration for task scheduling methodologies for space systems in the same way that the research in this thesis can potentially be applied to these non-space related domains.

4.0 Approach

The goal of the SMAARTE framework is to manage the execution of scheduled and routine tasks while appropriately handling unexpected events such as hardware and software malfunctions. To accomplish this goal, the Schedule Manager was developed, which is a collection of C/C++ functions to read and process schedule files containing tasks for execution. After developing the Schedule Manager, it was then integrated with cFE and PLEXIL to show an example of how it can integrate and work with existing software frameworks. Lastly, a Graphical User Interface (GUI) was created to enable a user to easily create schedule files for upload and to be able to send individual tasks to the system for simplified live testing.

4.1 Schedule Manager

The Schedule Manager was developed to define and autonomously manage the execution of tasks. To accomplish this, there needed to be a way to define tasks with varying types of constraints so the Schedule Manager would know how to properly handle unexpected issues without introducing new conflicts. The resulting definition of a task can be seen in Table 1 below.

Table 1 Schedule Manager Task Struct Fields

Field Name	Description
pid	The return of the fork function to the parent when launching the child executable
id	The identification number for the task, used to keep track of completed task entries
start_time	The start time of the task, earliest time the task can start
end_time	The end time of the task, latest time the task can start
duration	The expected duration of the task in seconds
conflict	The conflict categories for the task, each bit in base 2 corresponds to a category
priority	The priority number (higher is more priority), determines which tasks launch first
type	The type of the task (0 for executable, > 0 for functions)
interval	The number of seconds to add to the start and end time of a task upon execution
memory	The estimated number of bytes of RAM necessary to run the task
storage	The estimated number of bytes of storage necessary to run the task
power	The lowest battery percentage to be able to run the task feasibly
args	Arguments to add to the executable or function

The pid field is the only field in the task struct that is not provided by the user in the schedule. If the task is an executable, then the pid field is used by the Schedule Manager to keep track of the executed task process id and send/receive signals with it. The id field is used by the Schedule Manager when alerting the user of task activity or conflicts. The id becomes 0 when the task is finished executing and is ready to be removed, which allows the task to be overwritten by new ones. The start_time and end_time fields are used to define a valid time-window for the task

to start, with 0 signifying no restriction. The duration field is used to detect when a task is running longer than expected, which allows stalled tasks to be handled on a case-by-case basis. The conflict field is multi-purpose and enables the user to define categories that only one task can run in at a time. With each bit corresponding to a category (for a total of 32 categories in a 32-bit integer), the user can define categories to represent command subroutines (e.g. each command waits for the previous command to exit), resources (e.g. a camera or sensor), or task-level conflicts (e.g. movement tasks and image capture tasks). The priority field is used to determine which task starts first if multiple conflicting tasks are attempting to start at the same time (e.g. waiting for the same resource to free up). The type field is used to define if the task is an executable process or a function. Compared to executable processes such as image-processing, functions are for tasks such as clearing the schedule or removing specified tasks. The memory, storage, and power fields are example variable resource fields that can be used to check specific thresholds necessary for tasks to run. The memory, storage, and power fields are not fully implemented yet and are a goal for future work. Lastly, the arguments field is used to provide the task specifics such as the process or function to execute and any configuration options.

Two versions of the Schedule Manager were developed, one in C++ and then later one in C. First, the C++ Schedule Manager architecture will be described followed by the updated C architecture and reasoning behind the programming language transition.

4.1.1 Initial C++ Version of the Schedule Manager

The desired functionality of the SMAARTE framework required a way to keep a dynamic schedule in memory with constantly updating tasks and varying run status. With the eventual goal of being flown in space, the potential languages this could be written were either C or C++. Other

common options such as Python or Java were not feasible due to the difficulty of formal verification for code written in those languages. To simplify the process of adding, clearing, parsing, and printing tasks in a constantly updated schedule, C++ was chosen for the initial version of the Schedule Manager.

To receive tasks, a message queue interface was built for adding individual tasks and a schedule file interface was built for adding tasks in bulk. Using the stringstream feature, tasks are parsed directly into a task struct, which is then passed through a verify task function to make sure all fields were within acceptable bounds before being added to the schedule.

After an initial schedule is loaded in the Schedule Manager, the main processing loop begins. First, all active processes (tasks with pid > 0) are checked using a non-blocking waitpid call and, if the task has returned, it checks the exit status and handles it accordingly. If the task has exited, a cleanup function is called, which either adds the task's vector position to a queue of finished task positions or clears the pid field and modifies the start and end time parameters if the task is to be restarted or is a routine task.

Once the finished active processes are handled and any expired pending tasks (current time past the end_time field) are removed, the remaining active processes are checked and the Schedule Manager updates an active conflict array, where each index corresponds to a bit in the conflict field, and the entry in the array is the id of the task contributing to that conflict bit. After creating this array, a similar pending conflict array is then created for pending tasks that are ready to start. When filling the pending conflict array, entries are overwritten if a new task with the same conflict number has a higher priority.

The schedule is then processed for all pending tasks. If the task is ready to start, then the task is executed if the active conflict array does not have an entry at the index of the current task's

conflict field and if the current task's priority is equal to the priority in the index of the pending conflict array. After execution, the active conflict array is updated to prevent later conflicting tasks from running. For tasks with the same conflict and priority, the one higher up in the schedule is executed.

4.1.2 Optimized C Version of the Schedule Manager

Although the C++ version of the Schedule Manager used dynamic containers such as vectors and queues which could grow or shrink on demand, this flexibility was not a desirable feature for a reliable software architecture. When formally verifying flight software for space missions, static memory allocation is always preferred over dynamic memory allocation to reduce the possibility of memory leaks. The C++ version may be a better fit for other use cases like general-purpose use and to add more complex functionality that makes use of C++'s enhanced features. However, without needing the dynamic data types offered by C++ and in addition, with the goal of integration with cFE which uses C, a new version of the Schedule Manager was written in C for integration convenience and to simplify the code complexity.

The changes necessary included minor conversions such adding a task parser that added to a task struct one field at a time and changing the schedule from a vector of tasks to a static-length array of tasks. An unexpected result of this conversion however was that many initially complex structures were more naturally simplified. For example, most of the functions that initially required vector and queue pointers as arguments were able to be converted to using index arguments which is both simpler and safer. Also, with the schedule data structure being of static length, the idea to sort the tasks in the schedule based upon priority became apparent and seemed practical without the fear of exponentially increasing sort time overhead. This optimization uses C's built-in qsort

function from stdlib to sort every task in the array based upon the priority field, which helped with eliminating the need for the complex active and pending conflict arrays. The sorted schedule array can then be iterated through from the top, with pending tasks executed immediately when feasible and using a single conflict variable with bit masking to represent the currently used conflict categories. A simplified explanation of this optimization is described with an example scenario in Figure 2 below.

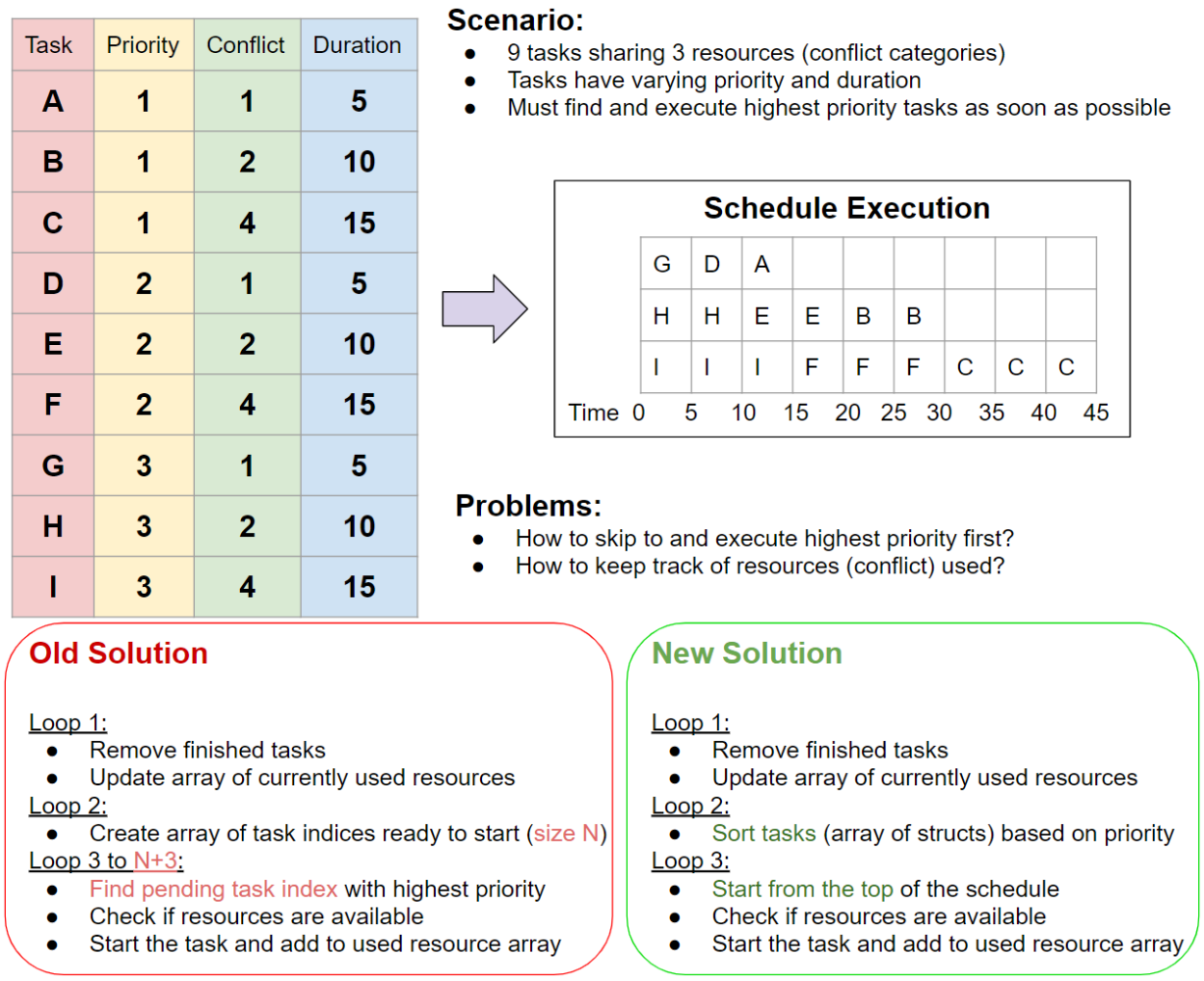

Figure 2 Schedule Execution Example

4.2 cFE, Schedule Manager, and PLEXIL Integration

The first integration scenario demonstrated during this project's development was a combination of cFE, PLEXIL, and the Schedule Manager. In this setup, the Schedule Manager was embedded in PLEXIL, and cFE would launch and communicate with PLEXIL in an external process.

There are four main components in this initial architecture: (1) the lightweight C++ library of schedule managing functions (Schedule Manager); (2) the PLEXIL Plan consisting of PLEXIL nodes; (3) the PLEXIL Adapter which acts as the interface between the PLEXIL Plan and the rest of the system; and (4) the cFS application (core Flight System, referring to non-core cFE applications) which launches PLEXIL and interacts with the rest of the cFE/cFS system applications. Figure 3 displays all four components together in an architecture diagram with cFE/ES (Executive Services) and cFS/HS (Health Services) as example cFE/cFS applications in the system.

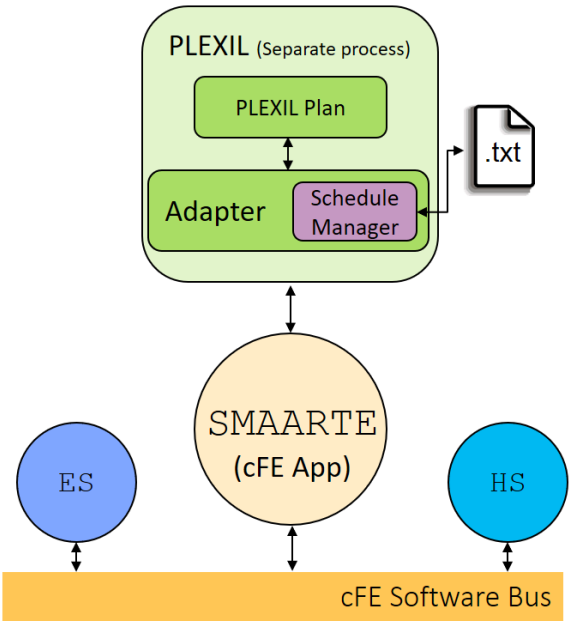

Figure 3 SMAARTE Architecture Diagram

In this framework, tasks can be PLEXIL adapter function calls, cFE/cFS commands (launched by sending a packet to cFE's command ingest), or process execution commands. These tasks would be input to and controlled by the Schedule Manager component and can be added to the schedule by sending commands to the cFS app via the command ingest, or by reading directly from a schedule file. The task schedule is routinely processed on an interval dictated by the PLEXIL plan and when tasks are executed, they can either execute external processes directly or trigger PLEXIL nodes.

4.2.1 PLEXIL Overview

A PLEXIL plan consists of a tree of varying types of nodes and enables deterministic execution. Each node has its own state and is connected to other nodes using one of several types of transitions available. The PLEXIL plan uses separate internal variables for its functionality, such as node conditions, so it needs to interact with the system using the PLEXIL adapter. The interface to this adapter consists of either Commands or Lookups. Commands can support a variable number of arguments and Lookups retrieve variables initialized as Lookup variables in the adapter. Nodes in PLEXIL plan files other than the main plan can be used with the LibraryCall utility.

PLEXIL nodes can be triggered externally by setting external variables (accessed with Lookups) as the start conditions for the nodes. In this framework, nodes that should be controlled by the Schedule Manager need two external variables in their start condition: an execute boolean and an end time. Execute is set to 1 when the node needs to be executed and is reset to 0 when processing the node. The end time variable is used to reset the execute variable to 0 if the node's

other start conditions prevent the node from running before the end time is reached. This method improves the flexibility of PLEXIL nodes and reduces the complexity of PLEXIL node start conditions. Without this method, for a given task that has a specific set of viable time-windows, either a PLEXIL node would need to exist for each window or a single node would need to have all the windows as conditionals. This method enables the Schedule Manager to maintain the conditions and simplifies the number of attributes needed as Lookups in PLEXIL.

The PLEXIL Adapter is the interface for PLEXIL to communicate with the rest of the system. The adapter is written in C++ and can interact with the system, as well as use the functions provided by the Schedule Manager to manage its own internal schedule. During adapter initialization, all Commands and Lookups are globally registered and then the base schedule is read from a file using the Schedule Manager functions. A message queue is then created for data to be input into the PLEXIL process from external processes (either cFE/cFS or other sources).

The integration of the Schedule Manager into PLEXIL demonstrated the ability of the Schedule Manager to integrate with and enhance the functionality of an existing plan execution framework, but this ended up not being a true integration with cFE. With the only interface between cFE and PLEXIL being a message queue interface, it would be inefficient for the Schedule Manager to interact with other cFE/cFS applications due to not having direct access to the cFE Software Bus (SB). To better demonstrate integration with cFE, a new system architecture was developed focusing on the Schedule Manager and cFE.

4.3 cFE and Schedule Manager Integration

The focus of this new integration scenario was to show how the Schedule Manager could be integrated directly into a cFS app to enable it to directly communicate on the cFE SB and consequently other spacecraft nodes as well via the SBN or SBD as mentioned earlier.

Using the C version of the Schedule Manager, integration with a cFS application was achieved by adding the appropriate Schedule Manager functions into the initialization (InitApp), run loop (AppMain), and command code (ProcessNewAppCmds) sections of a cFS application template. The result of this integration was a cFS application that can read input from multiple sources and call functions during runtime to enable full control of the Schedule Manager functionality.

Although this method of integration makes it more difficult to interact with parameters of external planning-languages such as PLEXIL, the ability to directly send and receive messages on the cFE SB makes it more responsive when needing to control the execution of cFE/cFS applications. Using the “type” variable in the task struct to switch from process-based execution to function-based execution, cFE/cFS applications could be managed by calling pre-written functions in the cFS application to send “start application” commands and receive application status updates using the cFE SB. This provides a responsive method to managing the execution of cFS applications. To retain the same flexibility as process-based task execution where the actual process call can be uploaded in the task argument field, a function could be used in cFE that reads the task arguments for the desired cFE/cFS application ID, command code, and arguments. To periodically check on a task, a routine task could be built in a similar manner. This integration is discussed further in the future work section below.

4.4 Python Tkinter Commanding GUI

With the enhanced flexibility and dynamic configuration capability of schedules created using the task structure provided by the Schedule Manager, it was important to have a feasible method of commanding and creating schedules for the system. The desired solution was to build a simple tool to enable a user to quickly create schedules using predefined templates. After a brief comparison of GUI options, Python Tkinter was chosen due to many benefits including being lightweight, portable, and either installed by default with Python or quick to install [18]. The GUI and example schedule can be seen in Figure 4.

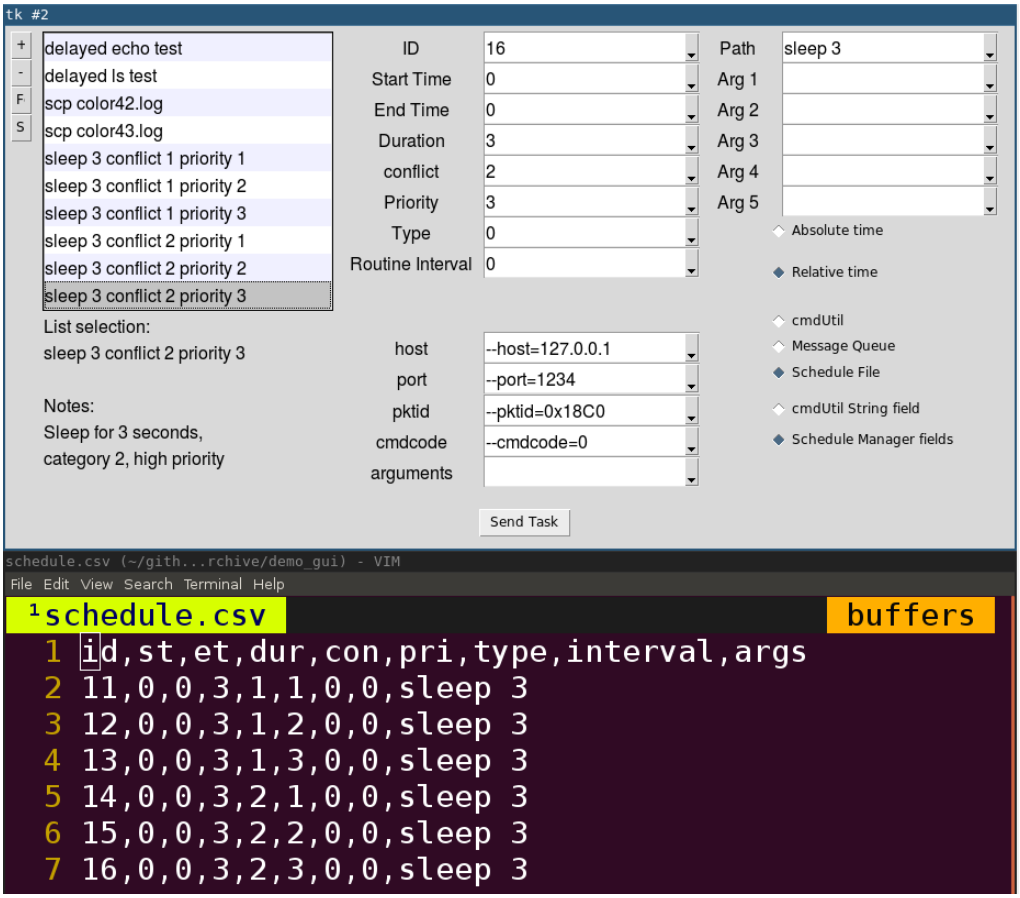

Figure 4 Python Tkinter Commanding GUI

In a custom GUI configuration file, commonly used tasks can be added to the scrollable list of preconfigured commands seen in the top left box in Figure 4. Clicking on one of these entries will populate the descriptions, input fields, and radio button options to the default set values. Before sending the task, modifications can be made by either clicking on the drop-down menu for common modifications or typing in values to override the defaults. Clicking "Send Task" will send the task to a schedule file by default unless another destination is selected using the radio button options. The Schedule Manager supports both .txt and .csv files. Compared to the TXT format, the CSV file format has a field description line and comma-separated values instead of space-separated values. The benefit of using CSV format is that the file can be directly opened and modified by spreadsheet editing tools such as Microsoft Excel. After adding a set of desired tasks to the schedule file using the GUI, the user can then use a text editor of their choice to rapidly modify, copy/paste, or delete tasks in the schedule. Other iterations of the GUI with built-in task swap/removal tools were also developed but using a familiar text editor often ended up being the most optimal method of schedule modification anyway due to the ability to change arbitrary task fields and being able to copy/delete arbitrary tasks in bulk with text find/regex tools.

Aside from assisting the user with creating schedule files, the GUI can also be used to directly send commands to running instances of the Schedule Manager, using a message queue if it is running stand-alone, or cmdUtil (command utility packaged with cFE) if run as a cFS application. Being able to send individual task commands in real-time enables the user to test and debug various features of the SMAARTE framework such as the start and end time fields.

5.0 Demonstrations and Results

This section will focus on how the developed software framework described in the Approach section can apply to real-world scenarios and enable or improve the management and execution of systems and missions. Due to the nature of this research, like others in the same field, this section will be more focused on demonstrating functionality and describing scenarios that this research enables rather than quantitative results and plots.

5.1 Schedule Execution Efficiency

Figure 2 above visualizes the execution of a simplified schedule with nine tasks sharing three resources. In this example scenario, the schedule is successfully executed after 45 seconds. For comparison, the same scenario with typical schedule execution restrictions is shown in Figure 5 below.

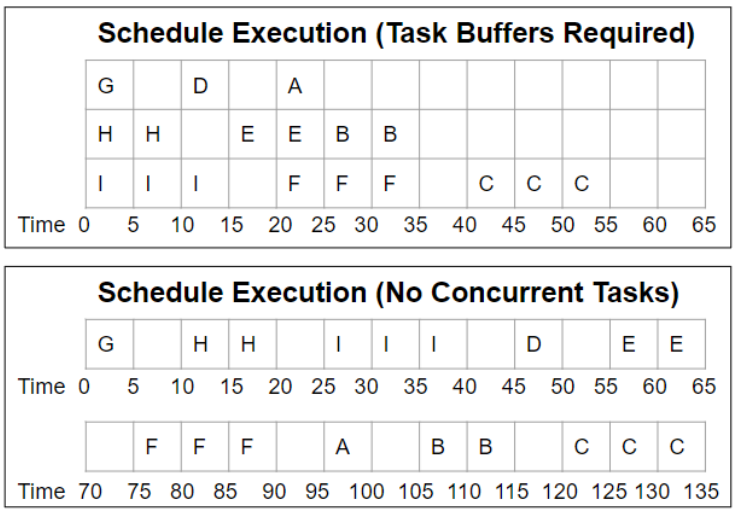

Figure 5 Schedule Execution with Restrictions

In typical task scheduling for remote mission execution, a buffer window is often scheduled between conflicting tasks. This approach is simpler for the schedule and the system architecture due to not needing the functionality to check if a task is finished, but there is the risk that a task can potentially run longer than the buffer window allows. Adding buffer windows with 33% the length of the longest expected task duration, the resulting schedule above shows that the same schedule can execute in 55 seconds. This result is a 22% increase in execution time compared to the original 45 seconds. However, in an even less sophisticated system where having separate task execution for every available resource is not built, then only one task would be scheduled to run at a time. In this case, the same schedule would need 135 seconds to execute, which is a 200% increase in execution time.

5.2 Example Schedule Manager Execution

The execution of the Schedule Manager can be monitored either by print messages in the terminal or by viewing and refreshing a file that contains the current schedule and status messages. During testing for small schedules, the simplest method is to see the print messages in the terminal, and this enables the user to quickly see the current state of the system. Figure 6 below shows the terminal output for the schedule shown in Figure 4 above.

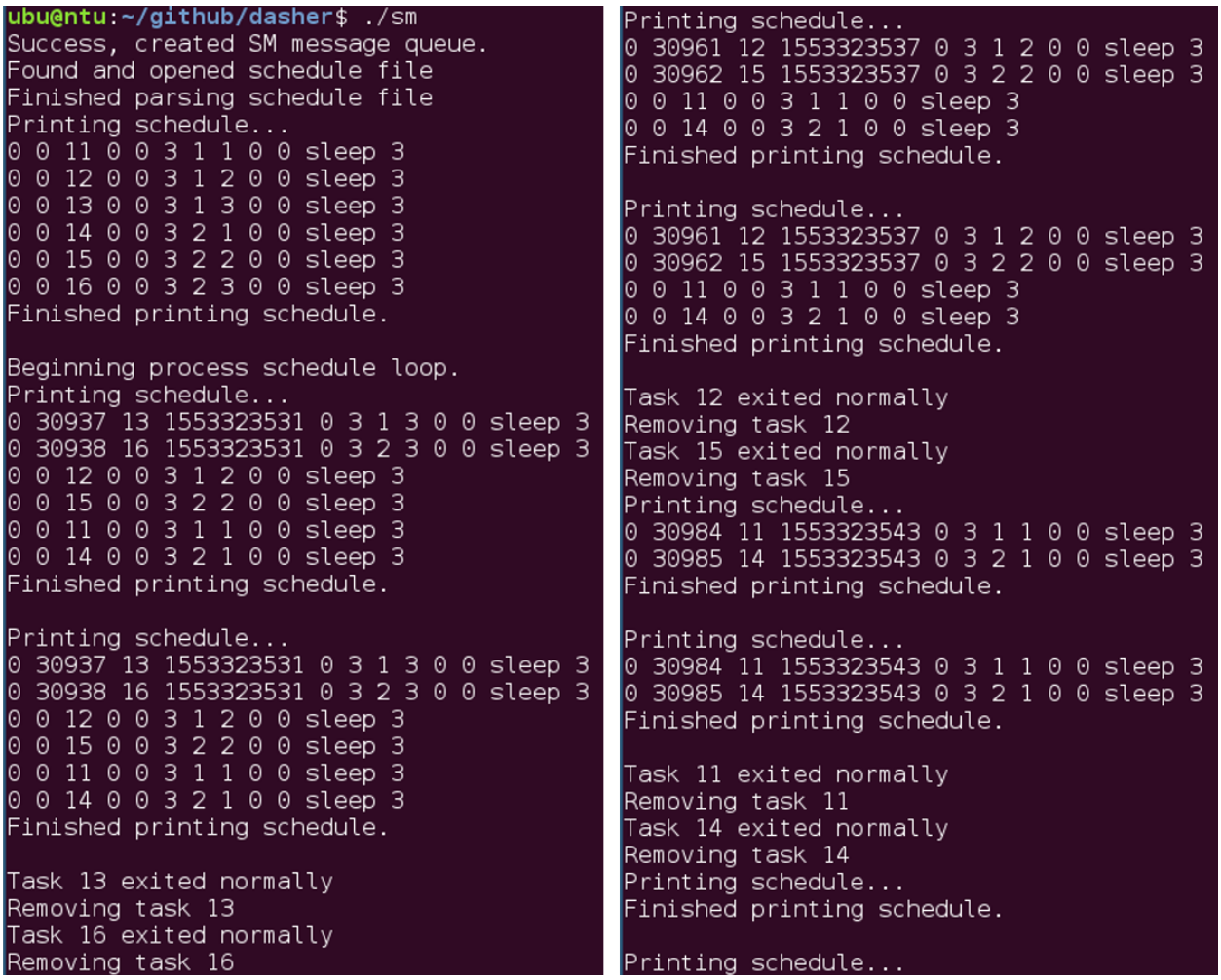

Figure 6 Schedule Manager Execution Terminal Output

This simple schedule execution highlights the main features of the Schedule Manager. After the initial load, the default task order is 11-16 with priorities in the order 1,2,3,1,2,3. After the process loop begins, the schedule is sorted based upon priority and the new order is 3,3,2,2,1,1. Also, the two highest priority tasks 13 and 16 with conflict fields 1 and 2 are executed, and the process id field is populated with the two process ids for the two sleep commands (30937 and 30938). The start time field is updated with the current system time which is used for the duration check. After task 13 and 16 are finished 3 seconds later, they are removed from the schedule and

the next two sleep commands are executed. This demonstration shows the core functioning principles of the Schedule Manager, and these are the main features leveraged to enable the demonstrations described in the later subsections.

5.3 Mission Management on STP-H6 (SSIVP)

As mentioned in the background section, the SSIVP experiment was developed at SHREC in preparation for the STP-H6 mission. SSIVP includes five main processors, two cameras, and one downlink, so proper mission management was necessary to make use of the system resources and downlink bandwidth.

Using SHREC's image processing library [19], images can be classified and compressed before downlink which can greatly improve the expected science return of the mission. For example, images can be classified by the Color Classifier app prior to downlink which uses RGB thresholds to estimate a percentage of clouds, water, land, and dark in a terrestrial-scene image, and images with a high land percentage can be prioritized in the downlink queue. Other applications like the Color Search app, which search for a specified range of RGB values, can also be used to define unique or high-quality images. Controlling the execution of these applications for routine downlink as well as being able to accept commands from the ground for image capture at specific times (for images of desired locations) requires the functionality provided by the Schedule Manager.

SSIVP was designed to be able to operate flexibly by enabling bash scripts to be uploaded and executed using a developed cFS application (SHL) which enables shell command execution. This removes the need to create new software OS images for every system modification and makes

it easier to design and upload new experiments. Because of this capability, the Schedule Manager can execute and manage bash scripts representing different mission goals. Figure 7 below shows two example image capture scripts, one to run routinely for rapid low-quality images, and one to run for single high-quality images.

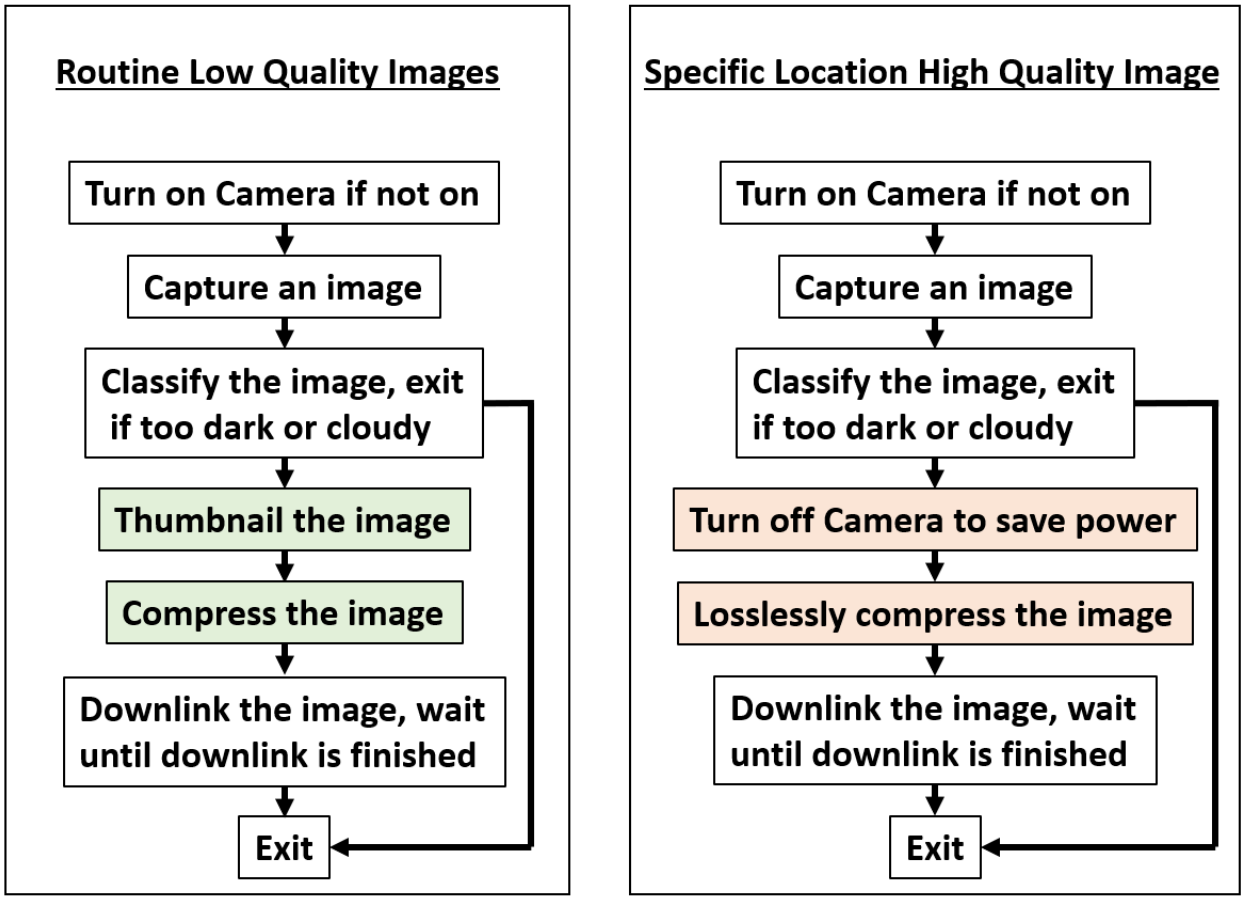

Figure 7 Example Image Capture Scripts

In the low-quality image script example, images are thumbnailed before compression to reduce memory consumption and speed up image compression. In the high-quality image script example, the camera is disabled to save power during the downlink (at least 20 minutes, more if the image is sent more than once to account for gaps in telemetry data).

Using the Schedule Manager, the low-quality image script could be a routine task that is run every minute. The high-quality image script would be an uploaded task that has the same conflict field and higher priority field compared to the low-quality image script, so that when the high-quality image script needs to run, then the low-quality image script will pause and wait for the high-quality image downlink to finish. This capability can be extended to support many other mission scenarios in script form and enables mission scripts to be added by third-parties without needing to redesign or remove currently running mission scenarios.

5.4 Mission Management on STP-H7 (CASPR)

The CASPR experiment is currently in the planning and design phase at SHREC for the STP-H7 mission. Although the exact details have not been decided yet regarding the type and quantity of processors and sensors for the experiment, it is clear that there will be more components than the power budget will allow. For example, it will not be feasible for real-time Neuromorphic camera processing, real-time Visual camera processing, and gimbal rotation to be operational at the same time. There is still adequate room in the budget however to run certain mission scenarios simultaneously.

Like SSIVP, CASPR will also support the running of science objective mission scripts. However, with the large quantity and variety of hardware and sensors, there will be many potential science objectives at any time. Although it is suitable to run one mission scenario at a time on SSIVP, the same method would be a waste of resources on CASPR. The Neuromorphic camera and Visual camera on CASPR for example could be controlled and used for two separate goals at the same time. In that case, instead of needing to think of every potential combination of science

scripts and manually combining them, multiple science objective scripts could run at the same time as long as the conflict and power fields are correctly set in the tasks.

As an example, the new SSP will be more power-hungry compared to the CSP and ranges from 4W to potentially 8-10W in power consumption depending upon the application. If running applications are lower in priority compared to capturing the images, then scripts with the sole purpose of collecting images from the Neuromorphic and Visual cameras would be executed instead of the image processing scripts when the power constraint is close to being reached. Also, more complex operations that will likely use up the entire budget such as capturing and processing images while the gimbal is moving can be set to conflict with any other potential mission script in the system.

Due to being connected to the ISS, instead of a restriction on the quantity of power used, the power constraint is in the form of a maximum instantaneous power draw (likely around 42W for CASPR). The functionality of the Schedule Manager along with the methodology described above will be suitable for the management of this type of experiment. In the case of a CubeSat experiment where the battery level also matters, a task to change operation modes (from operational to power-saving for example) would then also be needed. One method of doing this would be to have a task that blocks certain resources from being used.

5.5 PLEXIL and cFE Integration Demonstration

One significant development milestone of the SMAARTE framework was to provide the functionality necessary to enable formation-flying SmallSat missions on flight hardware such as the CSP. To achieve this goal, software was developed and tested on ZedBoards (FlatSat

development boards for the CSP flight computer as shown in Figure 8) which use the same SoC (Zynq-7020) as the CSP including a dual core ARM Cortex-A9 processor. Testing was performed on Ubuntu14.04 32-bit for the PC and both ArchLinux and Ubuntu16.04 32-bit for ARM. Multiple OS were used on the ZedBoards to identify potential portability concerns and issues.

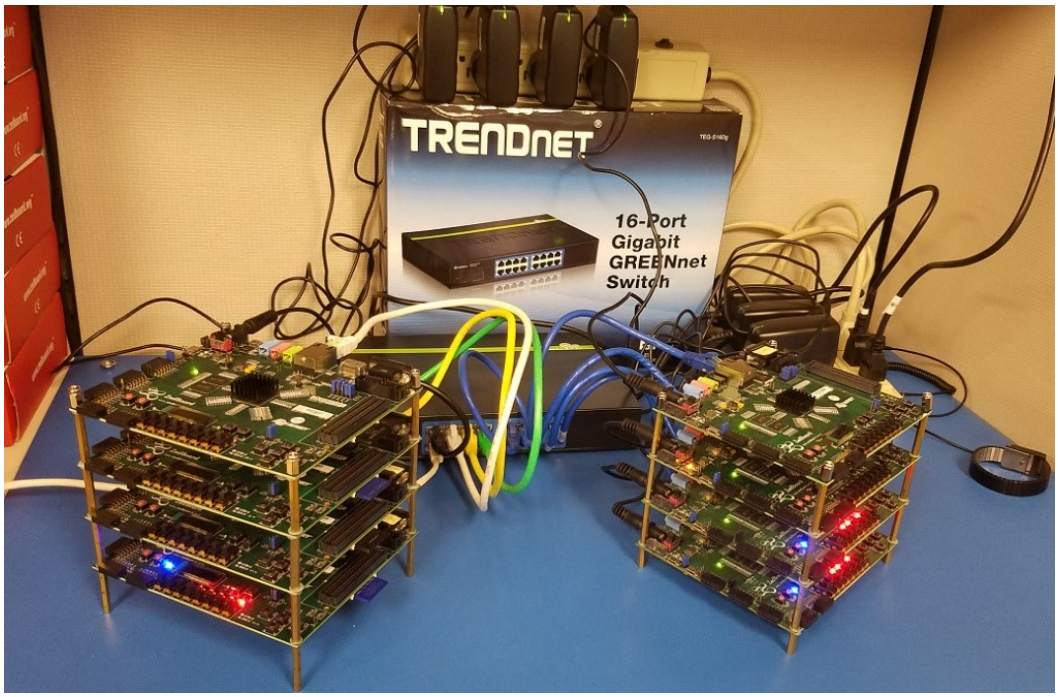

Figure 8 ZedBoard Cluster Testbed

For testing distributed computing with the SMAARTE framework, a cluster consisting of eight switch-connected ZedBoards was assembled. To facilitate testing and terminal management, most often only four ZedBoards were used for demonstrations (two with ArchLinux and two with Ubuntu).

5.5.1 Synchronized Event Demonstration with PLEXIL and cFE

A demonstration for the SMAARTE architecture was developed based upon a scenario focusing on event detection and synchronized response. In this demonstration, one of the boards randomly generated the detection of an event, representing a satellite detecting a forest fire. Once the detection event was generated, the board triggered a camera-synchronization event. For this demonstration, each board had the SMAARTE architecture loaded and running. The only notable difference was that the leader was monitoring a simulated camera to trigger the sending of tasks over the network.

The software architecture used for this demonstration is the one described in Section 4.2, with the Schedule Manager, PLEXIL, and cFE integrated together. In the demonstration, once a random-number generator reached above a set threshold, a value was set using the PLEXIL adapter to trigger the PLEXIL plan (through a Lookup) to call a PLEXIL Command to trigger a camera-synchronization event. The trigger for synchronization was routed through the PLEXIL plan to show the data-flow architecture for a system suitable for more complex situations. The arguments for the camera-event sync Command were set in the PLEXIL node to show the usage of a variable number of arguments in a PLEXIL node Command. Once the Command was received, the data was then sent to the other boards, where a camera event would be scheduled for a particular time-stamp (set to five seconds in the future). The result of this demo was that each board would print a task execution message at approximately the same time, five seconds later, with slight deviations due to the synchronized system time. In a fully developed system, network communication would be through a cFE/SB command being sent to the desired board via the SBN or SBD, but for demonstration simplicity communication was achieved using the cmdUtil tool packaged with cFE.

5.5.2 Schedule Manager Controlling cFE SB Demonstration

The demonstration described here is for the architecture described in Section 4.3, with just the Schedule Manager and cFE integrated. This integration however was straightforward, and the requirement for the demonstration to prove the framework was functional was to simply have the Schedule Manager start up, process a schedule, and forward a message over the cFE SB to another cFS application. The visual result of this demo was a brief cFE startup message followed by the output shown in Figure 6. When a task was sent using cFE's cmdUtil, the cFS app received it and printed the message, and then sent it to another cFS app (called the read app) using the cFE SB which printed the received message.

5.6 Hacksat and Pynq Board Schedule Manager Demonstration

To demonstrate live scheduled image processing and distributed communication, a demonstration was built using a Pynq board cluster (using the same Zynq-7020 SoC as the Zedboard) and Hacksat, as seen in Figure 9 below.

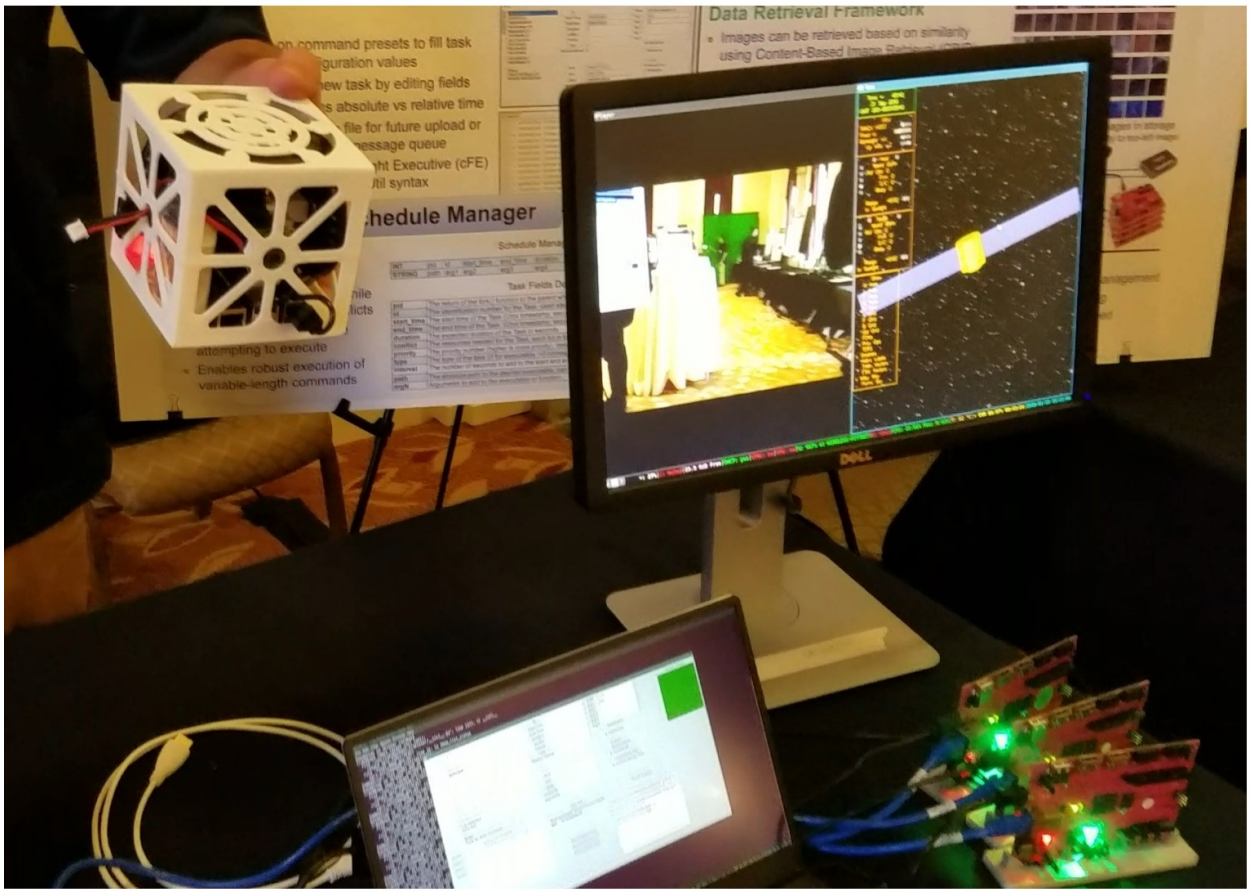

Figure 9 Hacksat and Pynq Cluster Demonstration

In this demonstration, the Hacksat represented a CubeSat doing image processing, and was built using a Raspberry Pi and Teensy among other components. It would stream a live video feed to the computer as well as send a periodic image capture to one of the Pynq boards. The Pynq board would then process the center pixel in the image to determine what color it was and then set its own RGB LED to that color. That board would then send the result to the other two Pynq boards and the ground station (Tkinter GUI) to display the color there. This information flow could be seen visually based upon the delay between the color setting of the coordinator Pynq board and

the ground station and worker Pynq boards. The Hacksat was also connected to a NASA 42 simulator instance [20] and the orientation data from the Teensy would be sent to the simulator so the simulation would match the rotation of the Hacksat in real time.

The image processing, LED color setting, and data transfer were all controlled by the Schedule Manager in this demo. Aside from demonstrating live routine task execution, this demo also demonstrated a new method to visualize spacecraft status on a development board using LEDs. To accomplish this, a custom bitstream was created in Vivado for the Pynq FPGA to connect the LEDs as well as buttons and switches to GPIO that the Linux OS could access. In a separate demo configuration, the buttons could be used to set the RGB LEDs to any color, and the switches could be used to disable commanding and/or telemetry functionality for that board to simulate communication failure. This method of live demonstration could potentially be used in the future to demonstrate complex distributed mission scenarios and simulate random external faults in an easier-to-observe manner than the same performed completely in simulation.

6.0 Discussion

The results and testbed demonstrations show that the SMAARTE framework enables the robust execution of tasks. By keeping track of active tasks to prevent conflicts with new tasks, task execution is not only safer, but also more reliable because this framework can detect and handle malfunctions with active tasks. This system, which is based upon dynamic task queues, enables the intentional scheduling of overlapping tasks, and can result in a more optimal solution than a static plan would be able to achieve, especially in scenarios where overlapping tasks finish earlier than expected. By not requiring a large safety margin to be scheduled between conflicting activities to account for uncertainty, the system resources can be more efficiently used, resulting in more efficient science data acquisition from the mission, as well as the ability to schedule tasks on the scale of seconds, not minutes. Additionally, although this system could benefit from onboard planning in terms of efficiency, it does not require onboard planning, because any deviation from the uploaded schedule is automatically handled by conflict checking and executing actions based upon priority. The main challenge with this system, as opposed to an onboard planning/scheduling approach as employed by CASPER, is when unexpected situations occur without a designed solution in the Schedule Manager. The spacecraft would then not be able to perform optimally, while a planning approach that considers initial state and goal state may be able to come up with an optimal solution for the mission.

7.0 Future Work

The current SMAARTE framework has some potential areas for further development and functionality enhancements. Many of these areas were cited in the sections above.

7.1 cFE/cFS Application Management

Although the Schedule Manager integration into a cFS app was completed and tested as described in Section 4.3, future work in this area includes finding or creating complex cFE/cFS systems and applying this research to those systems. Aside from designing the interface between the Schedule Manager cFS app and other cFS applications, extensive testing also needs to be performed to ensure that the software is safe for flight and can be used on more important missions. Creating unit tests and functional tests for the framework will be necessary to have this approach become adopted by others in the space community.

7.2 UAV Mission Management

This research has the potential to apply to other systems aside from space systems such as UAVs. The future work will be to investigate to what extent the Schedule Manager can enhance the functionality of autonomous UAV missions. With tasks and activities being executed at a much faster pace compared to space satellites, modifications may need to be made to the Schedule Manager to be more accurate and responsive for short time-windows. Also, depending upon the

requirements of UAV navigation and decision making, the Schedule Manager may either be able to control all the operations of a UAV, or just manage the execution of science objectives with a more traditional solution used for UAV navigation and control.

7.3 Distributed Swarm Control

The initial goal of the research was to enable distributed constellation and swarm missions which require autonomy to be manageable. The current framework already provides a potential solution for distributed scenarios with well-designed schedules on each vehicle in a similar manner to the demonstrations shown above. However, more work needs to be performed to provide examples of how more complex systems would be designed. One benefit of targeting UAV control with the Schedule Manager is that UAV swarm demos can be built and used to showcase the autonomous distributed system control in place of actual spacecraft clusters which are difficult to test with.

8.0 Conclusions

Adding autonomous capabilities to space systems provides many benefits, such as decreased ground-crew workload, increased efficiency, and more dependable systems. The approach presented in this thesis describes a relatively simplistic framework that provides another option to onboard autonomy aside from an onboard planner and scheduler.

This framework uses constraint and priority values associated with every task to create a system that can robustly execute tasks autonomously. Due to the conflicts between tasks being clearly defined in this system, the transition between sequential conflicting tasks can be handled instantly and automatically without requiring buffers to be scheduled between conflicting tasks. By having tasks relate to all other tasks based upon constraints and priority, unexpected delays are handled using the design of the system without requiring a separate phase of replanning and rescheduling.

Because of its integration with cFE and PLEXIL, the SMAARTE framework can be used to add functionality to an existing system already leveraging these platforms while reusing the original system's design components. This construction of an autonomous-capable system with open-source components has the potential to lead to increased collaboration between members of the space community and reduce the need to "reinvent the wheel" when it comes to development for new space missions. The flexibility of the stand-alone Schedule Manager and the demonstrations shown with it also work towards making it easier for others in the space industry to adopt this research for their mission needs.

Bibliography

[1] A. Gillette, B. O'Connor, C. Wilson, and A. George, "Spacecraft mission agent for autonomous robust task execution," in *2018 IEEE Aerospace Conference*, 2018, pp. 1-8: IEEE.

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

[3] J. Wilmot, "Implications of Responsive Space on the Flight Software Architecture," 2006.

[4] C. Manderino, A. Gillette, P. Gauvin, and A. D. George, "Resilient Networking Framework for Mission Operations and Resource Sharing in Multi-Agent Systems," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, 2018, pp. 1-7.

[5] T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso, and V. Verma, "Plan execution interchange language (PLEXIL)," 2006.

[6] D. Rudolph *et al.*, "CSP: A multifaceted hybrid architecture for space computing," 2014.

[7] C. Wilson *et al.*, "CSP hybrid space computing for STP-H5/ISEM on ISS," 2015.

[8] S. Sabogal *et al.*, "SSIVP: Spacecraft Supercomputing Experiment for STP-H6," 2017.

[9] S. Chien *et al.*, "Using autonomy flight software to improve science return on Earth Observing One," vol. 2, no. 4, pp. 196-216, 2005.

[10] K. Center *et al.*, "Improving Decision Support Systems Through Development of a Modular Autonomy Architecture," in *Proceedings of the 2012 ISAIRAS Conference, Turin, Italy*, 2012.

[11] C. Araguz, E. Bou-Balust, and E. Alarcón, "Applying autonomy to distributed satellite systems: Trends, challenges, and future prospects," *Systems Engineering,* vol. 21, no. 5, pp. 401-416, 2018.

[12] G. Beaumet, G. Verfaillie, and M.-C. Charmeau, "Feasibility of Autonomous Decision Making on Board an Agile Earth-Observing Satellite," *Computational Intelligence,* vol. 27, no. 1, pp. 123-139, 2011/02/01 2011.

[13] S. Spangelo, J. Cutler, K. Gilson, and A. Cohn, "Optimization-based scheduling for the single-satellite, multi-ground station communication problem," *Computers & Operations Research,* vol. 57, pp. 1-16, 2015/05/01/ 2015.

[14] D.-H. Cho, J.-H. Kim, H.-L. Choi, and J. Ahn, "Optimization-Based Scheduling Method for Agile Earth-Observing Satellite Constellation," *Journal of Aerospace Information Systems,* vol. 15, no. 11, pp. 611-626, 2018/11/01 2018.

[15] W. F. Boyer and G. S. Hura, "Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments," *Journal of Parallel and Distributed Computing,* vol. 65, no. 9, pp. 1035-1046, 2005/09/01/ 2005.

[16] S. Mahmoud. Zadeh, "Autonomous Reactive Mission Scheduling and Task-Path Planning Architecture for Autonomous Underwater Vehicle," *arXiv e-prints,* Accessed on: June 01, 2017. Available: https://ui.adsabs.harvard.edu/\#abs/2017arXiv170604189M

[17] Y. Khosiawan, Y. S. Park, I. Moon, J. Mukund Nilakantan, and I. Nielsen, "Task scheduling system for UAV operations in indoor environment," *arXiv e-prints,* Accessed on: April 01, 2016. Available: https://ui.adsabs.harvard.edu/\#abs/2016arXiv160406223K

[18] F. Lundh, "An introduction to tkinter," *URL: www. pythonware. com/library/tkinter/introduction/index. htm,* 1999.

[19] A. Gillette, C. Wilson, and A. D. George, "Efficient and autonomous processing and classification of images on small spacecraft," in *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, 2017, pp. 135-141.

[20] E. Stoneking, "An Open-Source Simulation Tool for Study and Design of Spacecraft Attitude Control Systems," ed, 2018, p. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20180000954.pdf.