Spacecraft Mission Agent for Autonomous Robust Task Execution

Antony Gillette[*], Brendan O'Connor[†], Christopher Wilson[*], Alan George[*]

[*]NSF SHREC Center, ECE Department,
University of Pittsburgh
4420 Bayard Street, Suite #560, Pittsburgh, PA, 15213
412-383-8142
{antony.gillette, christopher.wilson,
alan.george}@chrec.org

[†]Emergent Space Technologies
6411 Ivy Lane, Suite 303, Greenbelt, MD, 20770
301-345-1535
brendan.oconnor@emergentspace.com

Abstract—**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 by 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 allows for 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, allowing conflicting tasks to run sequentially, therefore 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

1. 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 [1]. 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 by handling any issues that arise without having to wait for instructions from the ground.

This paper 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 problems 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, allowing for flexibility due to 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). By using these constraints, the spacecraft schedule can be shifted earlier if allowed by the constraints, resulting in minimized delay between sequential tasks. Priority parameters allow tasks to be prioritized based on 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

978-1-5386-2014-4/18/$31.00 ©2018 IEEE

desired, and it also facilitates the addition and modification of tasks from the ground.

The SMAARTE framework was developed as a library, which allows it to be easily integrated into existing executives or simply run standalone. SMAARTE was developed and integrated with both 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 and priority functionality for other potential applications.

2. BACKGROUND

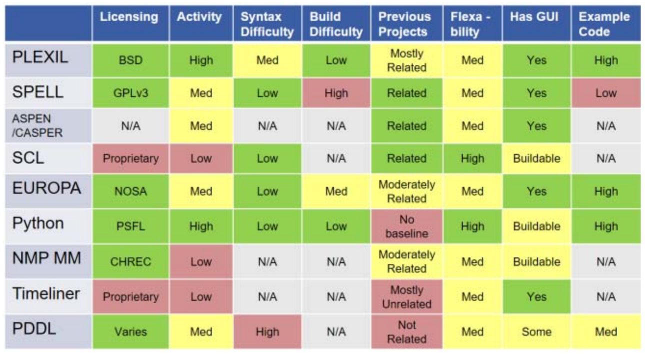
The main end goal for this research is to enable autonomous capabilities for distributed space missions (DSMs). The SMAARTE framework was built as a component in the Distributed Automation Suite for Heuristic Execution and Response (DASHER) project by Emergent Space Technologies in collaboration with 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.

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 [2] and wide community acceptance. 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) [3]. 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. Various applications are included with cFE that provide additional functionality such as a command ingest, telemetry output, and a software bus for cFE application communication. In addition, cFE works on top of the OSAL (Operating System Abstraction Layer), which allows cFE to work on Real-Time Operating Systems (RTOS) for missions requiring low jitter and determinism.

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 standalone platforms like the Robot Operating System (ROS) framework. Also, as the target platform is 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.

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 [4]. 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. More description and detail of PLEXIL can be found in [5].

3. 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.

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) [6]. 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.

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 [7]. 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 allow 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 allows 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.

4. 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. There are four main components of the architecture: (1) A lightweight C++ library of schedule managing functions (Schedule Manager); (2) PLEXIL Plan consisting of PLEXIL nodes; (3) PLEXIL Adapter which acts as the interface between the PLEXIL plan and the rest of the system; and (4) cFE application which launches PLEXIL and interacts with the rest of the cFE/cFS system applications. Figure 2 displays all four components together in a software diagram with cFE/ES (Executive Services) and cFS/HS (Health Services) as example applications in the system.

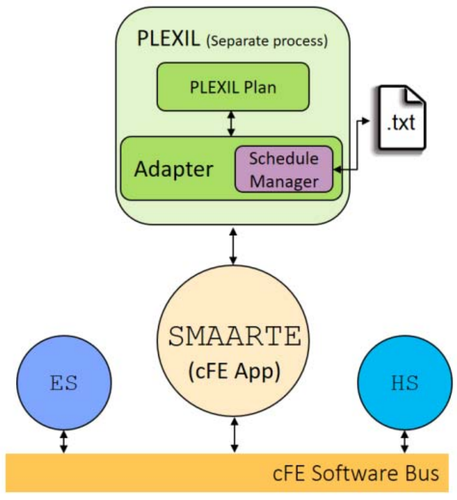


Figure 2. SMAARTE Architecture Diagram

The main operation of the framework relies on tasks with a standardized format. Tasks can be PLEXIL adapter function calls, cFE commands, or triggers for external applications. All tasks include key parameters, such as time constraints (start/end time), priority levels, and resource requirements. These tasks are stored in a dynamically sized schedule, which is controlled by the Schedule Manager component. Tasks can be added or removed from the schedule by sending commands to the cFE app via the Software Bus (which can interact with a cluster through the Software Bus Network, cFS/SBN), or by reading directly from a file. The task schedule is routinely processed on an interval dictated by the PLEXIL plan and, when the current system time passes the start time parameter, the task is either executed or put on hold if its required resources are not yet available. After task execution, the task is either removed, or a new start time parameter is generated if the task is configured as a routine task. The following sections describe each of the four main components of the SMAARTE framework in greater detail.

Schedule Manager

The main component of the SMAARTE framework is the Schedule Manager. The Schedule Manager allows for the creation and management of a dynamically sized vector of tasks. As mentioned previously, each task has its own table of key parameters (Table 1). Each task is individually initialized by providing the fields in white. The `pid` field (in grey) is added when the task executes, and the task is only removed from the schedule when the task ends. Currently the system can only monitor external processes, but a future goal is to integrate PLEXIL node and cFE application monitoring functionality. Table 2 provides a more detailed description for each field.

Table 1. Schedule Manager Task Fields

Type: int	pid	id	start_time
Type: int	end_time	duration	conflict
Type: int	priority	type	routine
Type: string	path	arg1	arg2
Type: string	arg3	arg4	arg5

Table 2. Schedule Manager Field Descriptions

pid	The return of the fork function to the parent when launching the child executable
id	The identification number for the Task, also used to keep track of completed Task entries
start_time	The start time of the Task (Unix timestamp, seconds since Epoch)
end_time	The end time of the Task (Unix timestamp, seconds since Epoch)
duration	The expected duration of the Task in seconds
conflict	The resources needed for the Task; Tasks with the same conflict number cannot run together
priority	The priority number (higher is more priority), it determines which Tasks launch first
type	The type of the Task (0 for executable, > 0 corresponds with application-specific functions)
routine_ interval	The number of seconds to add to the start and end time of a Task upon execution
path	The path to the desired executable, can be blank if type > 0
argN	Arguments to add to the executable or function

Task schedule processing is routinely triggered by a PLEXIL node in the PLEXIL plan. The PLEXIL node calls a PLEXIL adapter function, which uses the Schedule Manager functionality to process the schedule, as well as a queue for finished tasks. 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 the finished tasks queue or modifies the start and end time parameters if the task is to be restarted or is a routine task.

After all the ended active processes are handled, the remaining active processes are checked to monitor resource usage. The Schedule Manager updates a conflict array, where each index corresponds to the type of resource being used (one entry for every type of resource conflict possible), and the entry in the array is the priority value of the task using the resource. Typically, this resource would be a sensor (e.g. a camera) or an application acceleration module, which makes having one conflict per task suitable for many scenarios. An approach to handle more complex scenarios including multiple categories and variable resources is discussed in the future work section. After creating this array, a separate conflict array is created for non-active tasks (tasks yet to start with `pid=0`). When filling this array, entries are overwritten if a new task with the same conflict number has a higher priority.

The schedule is then processed for all non-active tasks. If the current time is past the `end_time` field, then it is too late to execute and the cleanup function is called. If the `start_time` field has passed but not `end_time`, 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 non-active conflict array (and the value is incremented so another non-active task with the same conflict and priority values will not also run). If the task cannot run, then it is put on hold and will keep attempting to run until the task's end time is reached. The driving logic behind this method is that, if there is a conflict, the priority entry will be the task's priority, if it is the highest priority task in the schedule for that conflict. For conflicting tasks with the same priority, the one higher up in the schedule is executed.

Tasks can be added one at a time through a message queue or in bulk by reading from an uploaded file. For receiving single tasks through a message queue, a task constructor is called with the arguments received. For reading from a file, a Schedule Manager function takes a text file with human-readable, space-separated entries as an input and adds each entry to the schedule.

When functions are called instead of executables (`type` field > 0), then in the PLEXIL adapter, a modified form of the task execution function is used where, instead of forking a new process, a function is called instead depending upon the type number. At any time, the current schedule can be printed using a Schedule Manager `print` function, which iterates through the vector of tasks and prints out all the fields space separated.

PLEXIL Plan

A PLEXIL plan consists of a tree of varying types of nodes and allows for 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 the SMAARTE 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 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 is beneficial because it adds functionality and can reduce the complexity of PLEXIL node start conditions. 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 allows the Schedule Manager to maintain the conditions in a simple table and simplifies the number of attributes needed as Lookups in PLEXIL.

PLEXIL Adapter

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 (combination of PLEXIL plan and PLEXIL adapter) from external processes (either cFE or other sources).

SMAARTE cFE Application

cFE is the baseline flight software for the system. Therefore, PLEXIL is launched from the SMAARTE cFE application, and all communication options are controlled by cFE commands.

The communication between PLEXIL and cFE is accomplished using a message queue, which is also the communication method of choice within the cFE framework. This approach was preferable to using semaphores, locks, or sockets because those methods would either introduce blocking, or result in potentially non-deterministic outcomes for the system.

5. RESULTS

To help convey the functionality of the SMAARTE framework, two main demonstrations were created. The demonstrations were tested both on PC and on development boards.

Hardware Configuration

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 CSPv1 [8]. To achieve this goal, software was developed and tested on ZedBoards (FlatSat development boards for the CSPv1 flight computer as shown in Figure 3) which use the same SoC (Zynq-7020) as the CSPv1 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.

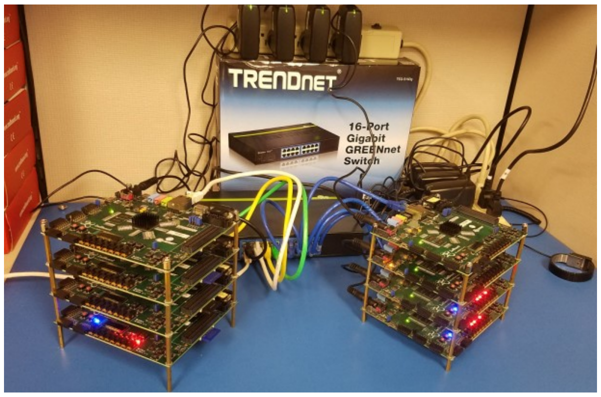


Figure 3. ZedBoard Cluster Testbed

For testing distributed computing with the SMAARTE framework, a cluster of eight switch-connected ZedBoards was assembled. However, for testing convenience and terminal management, mainly four ZedBoards were used for all demonstrations (two with ArchLinux and two with Ubuntu).

Synchronized Event Demonstration

A demonstration for the SMAARTE architecture was developed based on a use case focusing on event detection and synchronized response. In this demonstration, one of the boards randomly generated the detection of an event (such as a satellite detecting a forest fire). Once the detection event was generated, the board triggered a camera-synchronization event (with simulated cameras in the demo). 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.

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.

Priority and Conflict Demonstration

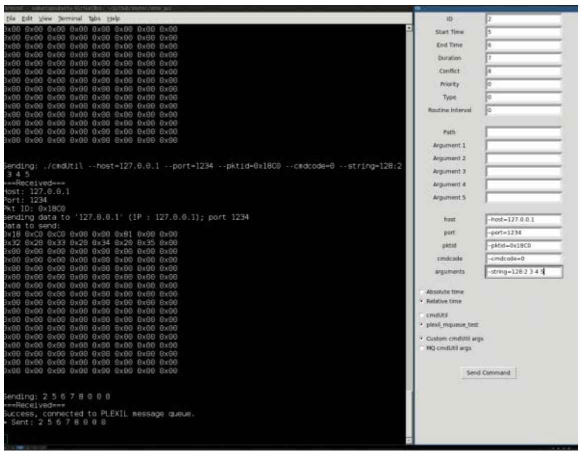
For the demonstration of priority and conflict functionality, various small, single-board use cases were developed. For testing purposes, the `sleep` system command was used to simulate long-running processes. To test priority, multiple sleep commands with the same conflict field and varying priority fields were added to the schedule with overlapping time constraints (specifically, three tasks were added with priority 1, 2, 3, and conflict 1), and it was shown that tasks would run in the order expected (3 then 2 then 1), with lower-priority tasks waiting for higher-priority tasks to finish. The same setup was tested with routine tasks instead and it was shown that, in the current system, conflict avoidance is functional but priority is only considered if multiple conflicting tasks need to start in the same schedule processing cycle. For priority to matter aside from this case, higher-priority tasks would need to be able to replace the lower-priority tasks currently running. This is described further in the future work section.

Demonstration GUI

A commanding GUI (Figure 4) was built for the demo using Python Tkinter, which is a lightweight and de-facto standard for efficient Python GUI development [9]. As opposed to the more commonly used Qt-based GUI development variants, Tkinter requires minimal setup and is either included in the default Python installation or is readily available in most package managers.

When the GUI initializes, default values are preloaded for the integer arguments and `cmdUtil` arguments to simplify the process of sending custom commands. There are 3 sets of radial buttons to select before sending a command. The first set is for absolute/relative time, which determines if the input `start_time/end_time` arguments require the current time to be added to it (i.e. if the user wants to execute a task 30 seconds later without calculating the current time + 30). The second set determines which executable to launch. If `cmdUtil` is selected, the `cmdUtil` arguments and either the task arguments or a set of custom arguments are sent (dependent on the third radial button set) and, if the PLEXIL message queue option is selected, then only the message queue arguments are sent using a message queue client executable for testing.

To send commands to other boards on the network, the only field that needs to be changed is the host field, and `cmdUtil` will automatically route the command to the target assuming there are no issues with the cluster network configuration.



```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00


Sending: ./cmdUtil --host=127.0.0.1 --port=1234 --pktid=0x18C0 --cmdcode=0 --string=128:2
 3 4 5
===Received===
Host: 127.0.0.1
Port: 1234
Pkt ID: 0x18C0
sending data to '127.0.0.1' (IP : 127.0.0.1); port 1234
Data to send:
0x18 0xC0 0xC0 0x00 0x00 0x81 0x00 0x00
0x32 0x20 0x33 0x20 0x34 0x20 0x35 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00


Sending: 2 5 6 7 8 0 0 0
===Received===
Success, connected to PLEXIL message queue.
> Sent: 2 5 6 7 8 0 0 0
```

ID	2
Start Time	5
End Time	6
Duration	7
Conflict	8
Priority	0
Type	0
Routine Interval	0
Path	
Argument 1	
Argument 2	
Argument 3	
Argument 4	
Argument 5	
host	--host=127.0.0.1
port	--port=1234
pktid	--pktid=0x18C0
cmdcode	--cmdcode=0
arguments	--string=128:2 3 4 5

Absolute time [✓ Relative time]

cmdUtil [✓ plexil_mqueue_test]

[✓ Custom cmdUtil args] MQ cmdUtil args

Figure 4. Python Tkinter Commanding GUI

6. DISCUSSION

The results and testbed demonstrations show that the SMAARTE framework allows for 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 on tasks waiting on active tasks, allows for the intentional scheduling of overlapping tasks, and can result in a more optimal solution than a planner/scheduler approach 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 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 on 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 need to default to a safe mode, while a planning approach that considers initial state and goal state may be able to come up with a solution to achieve the mission goals.

7. FUTURE WORK

The current SMAARTE framework has many potential areas for further development and functionality enhancements. Some of these enhancements would be simple to develop yet powerful additions.

Resource Availability

In the current system, every task is limited to one type of resource (such as a sensor) due to the current design of the conflict field. A potential solution for representing usage of multiple resource categories is to embed the categories into a base-2 number. For example, in a system with four sensors, the conflict field of a task requiring the 1st and 4th sensors would be $1001_2 = 9$. This system would allow for the representation of 32 distinct categories of resources, constrained by the size of an `unsigned int`.

For the representation of variable resources (such as memory or power), the solution would be to add additional conflict fields for each variable resource representing the maximum possible usage of that resource for the task. An additional check would then be developed to determine the available system resources before running the task.

Priority Overrides

In the case where a high-priority task needs to run but conflicts with currently running tasks, the lower-priority tasks should be preempted (checkpointed or killed), and potentially restarted afterwards depending upon the situation. A common use case of this is recovery modes. If the system is running out of memory or power, this functionality would allow for the currently active tasks to be processed to analyze resource usage and recover them appropriately. Another example is if sudden maneuvering is necessary, all the sensors and all processes using them can be disabled until the maneuver is complete.

Malfunction Handling

Currently, the schedule manager functions can check process-signal returns and detect processes running longer than estimated, but the method to handle these cases appropriately needs to be developed. Similar to the previous example, tasks should likely be killed and potentially restarted if anything abnormal is detected.

Schedule Analysis

When a schedule is created, due to the task structure including time constraints and estimated duration, it is possible to analyze if any tasks would be unable to run due to conflicts. Problems with the schedule can be determined by simulating the execution of the Schedule Manager on a faster time scale. Instead of polling the system time, a simulated time would be polled instead, and all commands would be replaced with a sleep command (which would need to use the same simulated time scale). Using this method, any issues such as a task not being able to run before its end time, or two routine tasks conflicting at regular intervals, would be discovered by checking the resulting logs.

8. CONCLUSION

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 paper 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 large buffers between conflicting tasks. By having tasks relate to all other tasks based on constraints and priority, unexpected delays are handled based on 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 cFE or PLEXIL system 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.

ACKNOWLEDGEMENTS

This research was funded by industry and government members of the NSF SHREC Center, the National Science Foundation (NSF) and its I/UCRC Program under Grant Nos. IIP-1161022 and CNS-1738783, and NASA STTR contract NNX16CG21P. The authors thank NASA Goddard and NASA Ames for providing the open-source projects cFE and PLEXIL, respectively. In addition, the authors thank John McGreevy from Emergent Space Technologies for discussions and interactions on several topics.

REFERENCES

[1] Achieving Science with CubeSats: Thinking Inside the Box, Washington, D.C., USA: National Academy Press, 2000. [Online]. Available: https://www.nap.edu/catalog/23503/achieving-science-with-cubesats-thinking-inside-the-box

[2] J. Wilmot, "Implications of responsive space on the flight software architecture." Proc. of AIAA Responsive Space Conference, 2006.

[3] S. Sabogal et al., "SSIVP: Spacecraft Supercomputing Experiment for STP-H6," *31st Annu. AIAA/USU Conf. on Small Satellites*, Logan, UT, August 5-10, 2017.

[4] *PLEXIL Getting Started*. [Online]. Available: http://plexil.sourceforge.net/wiki/index.php/Main_Page

[5] T. Estlin et al., "Plan Execution Interchange Language (PLEXIL)," Moffett Field, CA, Tech. Rep. NASA/TM-2006-213483, Apr. 2006.

[6] S. Chien et al., "Using autonomy flight software to improve science return on Earth Observing One", J. Aerosp. Comput. Inf. Commun., vol. 2, no. 4, pp. 196-216, Apr. 2005.

[7] K. Center et al., "Improving Decision Support Systems Through Development of a Modular Autonomy Architecture", Proceedings of the 2012 I-SAIRAS Conference, Turin, Italy, September 2012

[8] D. Rudolf et al., "CSP: A Multifaceted Hybrid System for Space Computing," *Proc. of 28th Annual AIAA/USU Conference on Small Satellites*, Logan, UT, August 2-7, 2014

[9] *Tkinter – Python Wiki*. [Online]. Available: https://wiki.python.org/moin/TkInter

BIOGRAPHY

Antony Gillette *is a doctoral student in ECE at the University of Pittsburgh. He is a research assistant of the hybrid space computing group in the NSF SHREC Center at Pittsburgh. His research interests include autonomy, flight software, and image processing for space systems.*

Brendan O'Connor *is the Chief Software Engineer for Emergent Space Technologies. He focuses on developing high availability and autonomous software systems for space applications.*

Christopher Wilson *is a doctoral candidate in ECE at the University of Florida. He is a research assistant and team leader of the hybrid space computing group at the University of Pittsburgh in the NSF Center for Space, High-performance, and Resilient Computing (SHREC) which replaced the NSF Center for High-performance Reconfigurable Computing (CHREC). His research interests include fault-tolerant techniques on hybrid architectures and radiation effects on commercial devices.*

Alan D. George *is Department Chair and R&H Mickle Endowed Chair in Electrical and Computer Engineering in the Swanson School of Engineering at the University of Pittsburgh. He founded and directs the NSF Center for Space, High-performance, and Resilient Computing (SHREC), which replaced the NSF Center for High-performance Reconfigurable Computing (CHREC) in late 2017. Dr. George's research interests are in advanced architectures, apps, networks, services, systems, and missions for reconfigurable, parallel, distributed, and dependable computing. He is a Fellow of the IEEE.*