FSW DESIGN SIMULATION FRAMEWORK

Brandon Molyneaux

November 8, 2020

Written for the NASA CubeSat Program FSW Team

Table of Contents

1 Introduction

This package is a framework that allows flight software (FSW) designs to be simulated with ease. After creating software states, which is an abstraction of satellite software flow and encapsulates FSW processes, the user can utilize this framework to generate a meaningful output for their FSW design.

This paper documents the following (not in any particular order):

1. A black-box perspective of how the framework components are connected.
2. What the user should be editing inside of the framework.
3. How the framework is used to simulate FSW design packaged together with a demo.

It should be noted that there is a lot of terminology that is thrown around in this simulation. It also should be noted that the terminology from FSW design, simulation, and NASA's F Prime shouldn't be used interchangeably, as some definitions differ between them. Unless otherwise explicitly defined, definitions and terminology may be interchanged.

2 Motivation

If the FSW design utilizes NASA's F Prime open source framework [4], chances are the developers will collaborate with systems engineers in a system modeling software such as Cameo Enterprise Architecture. In this software, there is a simulation tool, but it does not allow the developer to get deep insights of the FSW design. Thus, this simulation was born.

This is where this simulation framework's power shines: it allows developers to run a statistical analysis (such as determining the number of times a certain state is visited) and develop data structures (trees, graphs) and run associated algorithms on the data structure(s) (such as Dijkstra [2], Kruskal [1], etc.) once the simulation runs. This post-simulation analysis will help identify FSW design fallacies, such as redundancies and non-optimized components of the FSW design (for example, continual checking to ensure a specific component of the satellite is on/off/functioning properly).

3 Framework Language and Library Dependency Choices

The choice of language for this framework is Python. There are a few reasons as to why this language was chosen over others, one of which is because this is the author's "go-to" language for most programming projects. The second, and more important reason, is because of the complexity of the framework; working in a high-level language and making it "as English as possible" will help flatten the learning curve for other developers of the framework and the users.

The author of this framework wanted this to be as close as possible to the standard library so that errors regarding local installations are minimized. However, storing FSW data (such as thresholds, hardware information, and algorithm initializations) would pose a challenge if developed using the standard library. A popular Python data handling library, Pandas [3] (version 1.14), was settled upon. The reason for this is because pandas makes data analysis easy through its tabular structure. It should be noted that data internally is structured either as a pandas DataFrame or a Series.

4 Framework Architecture

Understanding the architecture for the framework is critical to understand how all of the components are connected, and more so for those who wish to contribute to the framework. Figure 1 depicts the architecture of how the FSWS is structured with respect to what the user will edit when constructing their FSW design.

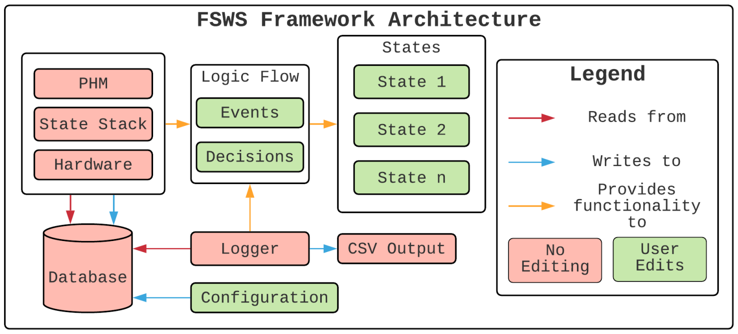


Figure 1 FSWS Framework Architecture

Before describing the architecture, it is important to note that all classes in this framework are referred to as a component and will be called a component hence forth. Thus, when discussion of inheritance begins, it may be slightly misleading that a component inherits a component; this is simply a class inheriting the methods and attributes of another class (OOP principle).

It is also important to note that part of this section of this documentation discusses the architecture from a *user perspective*. That is, editable components are components that the *user* will edit, and conversely the non-editable components are components the *user* won't edit. The user is able to *utilize* the non-editable components. Editing a non-editable component directly could cause the framework to not work properly and is highly discouraged. The user should not have to add in additional components as well from scratch. Any components that need to be created to add into the framework have a stub that the user can work from, which is described in the documentation on GitHub.

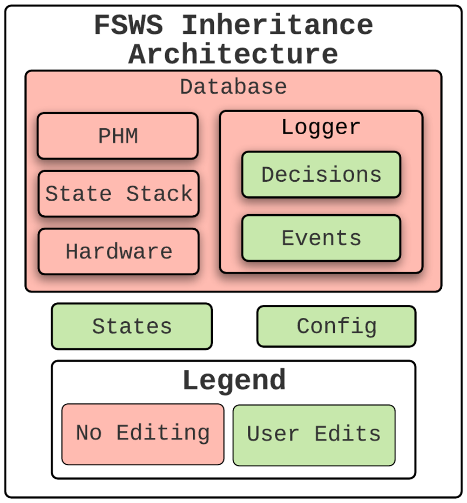


Figure 2 FSWS Inheritance Architecture

This framework uses inheritance to be able to pass data around during the simulation, and the inheritance architecture is shown in Figure 2. This is how the components "talk" to each other – the data is stored in attributes and methods provide the functionality to edit the attributes in the database.

Upon running the simulation, the configuration file is loaded into the database. The Logger component only reads from the database and the PHM, Stack, and Hardware components read and write to/from the database. From here, the user is able to utilize methods provided in these components to write the logic flow of what happens at a given software moment (box or diamond in the state diagrams). The states are the driver of the simulation; they utilize what the user wrote in the Events and Decisions components, which utilize Logger, PHM, Stack, and Hardware functionality, which then talk to the database to retrieve/store values. Once the simulation is finished, it will output a CSV file for the user to perform post-simulation analysis.

5 Components

The framework is comprised of components, which are individual parts that comprise a section of the framework with unique functionality. As mentioned in the last section, there are some components the user will edit, and some the user won't. There shouldn't be any need to add in additional components without using built-in functionality to stub out a specific component.

5.1 States

States encapsulate the software's logic flow. This component is essential to utilizing the framework; if the satellite design isn't broken up into states, then this framework cannot be used. These are the components of the FSW design that will be edited directly and will be the driver to making the simulation work properly. The states use what the user coded in the Events and Decisions components; it does not utilize anything in the configuration, logger, etc. components. The States component code is stubbed out; to create a new state, navigate into the states folder and type in terminal/command prompt `python3 stub.py` and then input the name of the state.

For example, if the user wanted to add in a communications state, the user would type in the states folder of the project directory `python3 stub.py comms`. A new file labeled `comms.py` will appear with portions of this stub that need to be edited. The portions where the user edits is clearly labeled with comments indicating where the user should put their code. This is found in the `phm_values()` and `run_process()` methods. Code Block 1 shows a newly stubbed state. At the time of writing, `__init__()` needs to be updated appropriately, as this was an old architectural design.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class SamplestateState:

    ''''Process code for Samplestate state.'''

    utils = Utils()

    def __init__(self, information):
        self.this_state = information['current_state']
        self.previous_state = information['previous_state']
        self.stack = information['stack']

        self.util_header = {'previous_state' : self.previous_state,
                            'stack_to_string' : self.stack.to_string(),
                            'current_state' : self.this_state,
                            }

    def phm_values(self):
        ''''Sets the PHM values for the Samplestate state.'''
        # DO NOT INCLUDE REACTION WHEEL SATURATION LIMITS! This is hardware
        # based, so they will be checked in the PHM.

        # === EDIT THIS DICTIONARY WITH PHM FLAGS FOR THIS STATE ===

        flags = {'state' : 'samplestate'}

        # ============================================================

        return flags

    def run_process(self):
        ''''Driver code to run a simulation for the Samplestate state.'''

        # print out end/start of new state.
        self.utils.header(self.util_header)

        # ==== YOUR CODE GOES HERE. ====




        # ==============================

        print("==== END SAMPLESTATE ====\n")
        return self.stack


# This is the test driver code. Be sure to comment/delete before integrating it
# into the driver code!

# import stack
# information = {'previous_state' : 'START!',
#        'stack' : stack.StateStack(), # just instantiate a new object for now.
#        'current_state' : 'samplestate'}

# information['previous_state'] = SamplestateState(information)
# information['previous_state'].run_process()
```

Code Block 1 Fresh State Stub

A sample of what goes inside of the `run_process()` method is shown in Code Block 3. This code block gives an example of how the state interacts with the Events and Decisions components.

5.2 Events and Decisions

Events and decisions provide the interface to the States components. They provide the *functionality* (or the *behavior*) of any given box or diamond in the software state diagram. A sample of this can be found in Code Blocks 4 and 5.

When looking at a state diagram, an *event* is any box except any superclass (thermal control, point at earth, etc.). The converse is a *decision*, which is a diamond in the state diagram. This diamond, when translated to code, is a decision of some sorts: either true or false.

It should be noted that every decision and event must include a string representation. This string representation is what will be shown in the final output. A full list of the attributes and methods for decisions and events can be found in the GitHub [link] repository documentation. Sample code for what the Events and Decisions components look like can be found in Section 6 of this document.

5.3 Hardware

This user-editable component consists of the hardware that's found on the satellite. The user does not need to edit the hardware file directly, as it provides the functionality to add into the simulation. Rather, every piece of hardware that is registered in the simulation has 2 forms: a

DCO for user interfacing and in the database (pandas DataFrame). More information about DCO's can be found in the Interfacing section of this document. The user can register (or add) the hardware in the configuration file as such:

```python
# Adds hardware to the simulation: Solar Panel A, computer and star tracker.
hardware.add('Solar Panel A')
hardware.add('Computer')
hardware.add('Star Tracker')
```

Code Block 2 Sample Hardware Registration in Configuration File

This will add in the hardware sub-component with 2 attributes (by default): `is_on` (boolean value) and `name` (string). More information on the attributes and methods regarding hardware can be found in the GitHub [link].

5.4 Configuration

This file configures the project. This is one of the only files that the user will edit that's directly built into the framework. It acts as the interface to set logger values, adding in hardware and hardware specifications, and setting general configuration parameters for the project.

At this current time, the configuration file is not fully implemented, and it is unclear as to what will be in this file exactly. Once this is figured out, this section will be updated. However, the general idea is that the user will be able to add in the various hardware, as well as registering the states in this file by using DCOs provided to the user (see section about interfacing).

5.5 Logger

The Logger is one of the biggest back-bone components to the simulation framework, as it is going to allow the users to be able to get meaningful results from the simulation. Conversely, this component was considerably the most difficult component to build and a considerable amount of time was put into developing the architecture of this component, which in turn caused design issues for the project within itself. Figure 2 depicts how the logger is integrated with the rest of the components.

The way that the user communicates with this component is through nodes, which is one row of the output. Before pushing the node to the logger officially, the logger will read what is in the database, add it onto the node, and then add it to the output DataFrame. At this current time, the only time that the nodes are utilized is when the user is developing the Events and Decisions components.

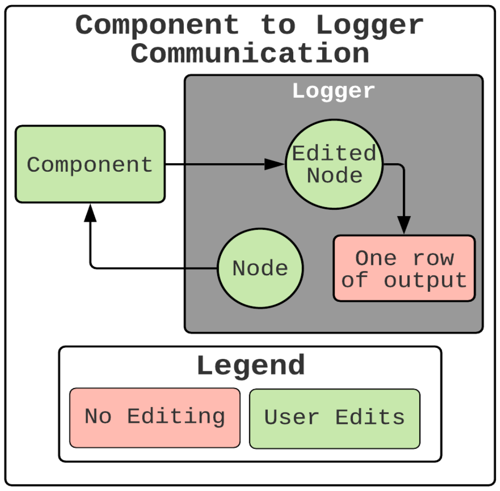


Figure 3 Component to Logger Communication

5.6 State Stack

As mentioned in the state component section, the way that this FSW simulation is built is by building the FSW design around an abstraction of procedures, which is better defined as states. The way that the transition from state to state occurs is through the state stack, which is built into the simulation tool. The functionality to transition to another state is already built into the events class, thus there is no need to build a method to transition states.

5.7 PHM

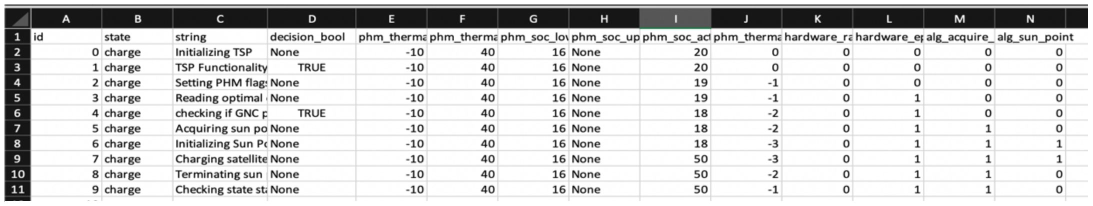
The Prognostic Health Management (PHM) system is used to monitor the satellite's health status (similar to how a doctor would monitor a human's blood pressure, weight, etc). This is built directly into the simulation and does not need editing. The values are set in the respective states, and when the simulation begins, it will read and store these values internally and be continually checking to ensure that randomly generated values during the simulation.

The satellites "health" is currently not simulated, but in a future update this should be looked into. The satellite's health should be updated after every time a new node is added into the logger.

5.8 Output

The output is handled by the logger superclass. Once the output is given to the user, the simulation will come to a halt. As of right now, the only output that the user will receive is a csv filed with the file format following *MMDDYY_HH:MM:SS_SIM*, where MMDDYY is the month, day, and year (respectively) of when the simulation was run based off of local time, and HH:MM:SS is hour, minute, and second the simulation was run based off of local time.

The output will vary from simulation to simulation depending upon the PHM values, what hardware was configured, etc. However, it will contain a string representation of the event that occurred (event or decision), an ID associated with the event, and all of the PHM values, what algorithms were used, and what hardware is turned off/on. Figure 4 shows an example of what this output looks like.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	id	state	string	decision_bool	phm_therma	phm_therma	phm_soc_lov	phm_soc_up	phm_soc_act	phm_therma	hardware_ra	hardware_ep	alg_acquire_	alg_sun_point
2	0	charge	Initializing TSP	None	-10	40	16	None	20	0	0	0	0	0
3	1	charge	TSP Functionality	TRUE	-10	40	16	None	20	0	0	0	0	0
4	2	charge	Setting PHM flag:	None	-10	40	16	None	19	-1	0	0	0	0
5	3	charge	Reading optimal	None	-10	40	16	None	19	-1	0	1	0	0
6	4	charge	checking if GNC p	TRUE	-10	40	16	None	18	-2	0	1	0	0
7	5	charge	Acquiring sun po	None	-10	40	16	None	18	-2	0	1	1	0
8	6	charge	Initializing Sun Pc	None	-10	40	16	None	18	-3	0	1	1	1
9	7	charge	Charging satellite	None	-10	40	16	None	50	-3	0	1	1	1
10	8	charge	Terminating sun	None	-10	40	16	None	50	-2	0	1	1	0
11	9	charge	Checking state st	None	-10	40	16	None	50	-1	0	1	1	0

Figure 5 Simulation output example

5.9 Example of State, Events and Decisions Components Relationships

This section details how a given State subcomponent and the Events and Decisions components are related through code. First, we must examine how the software flows at a high level through a software state diagram, depicted in Figure 6. Note that this is a *very* stripped-down version of what a software state would look like in practice. This is only shown in this documentation for demonstration so that the user is able to understand how to write proper code for the Events and Decisions components.

In this state, the satellite establishes the thermal set point (event), then verifies if it works properly (decision). Suppose that this check passes, the satellite will then set the PHM flags (event), read the optimal charge vector (event), then check to see if the peripherals are on (decision). If not, it'll turn the peripherals on (event) and check to make sure they're working (decision). If so, it'll acquire the sun position (event) and then point towards the sun (event) and charge (event). Once it's done charging, it'll terminate the sun pointing algorithm (event), then check the state queue for the next state (event).

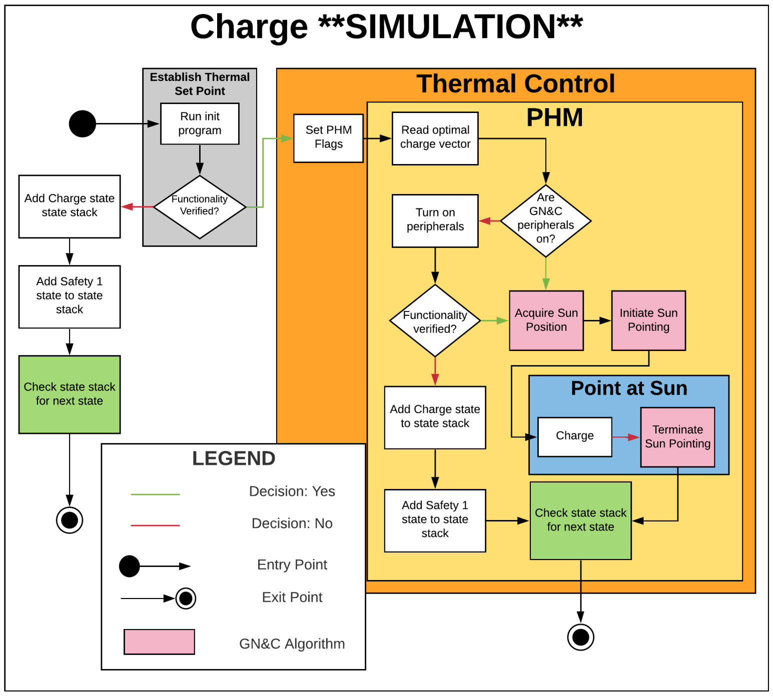


Figure 8 Charge Software State Diagram

All that's left is to do is build the Events and Decisions components first, then build out the Charge State sub-component. <u>**Note that the Events and Decisions components can get quite lengthy in terms of code length**</u>. Thus, Table 1 gives an overview of the methods that you would find in the events and decisions classes for the Charge State component. A fully stubbed out example of the events and decision classes is found in Section 7 of this document.

Events Class Methods	State Diagram Box Label
`add_state_to_state_stack`	Add {state_name} state to front of state queue (built-in)
`check_state_stack`	Check state stack for next state (built-in)
`set_phm_flags`	Set PHM flags (built-in)
`read_opt_charge_vector`	Read optimal charge vector
`charge_establish_tsp_init_program`	Run init program
`turn_peripherals_on`	Turn peripherals on
`alg_acquire_sun_position_init`	Acquire Sun Position
`alg_sun_pointing_init`	Initiate Sun Pointing
`alg_sun_pointing_terminate`	Terminate Sun Pointing
`charge`	Charge

Decision Class Methods	State Diagram Diamond Label
`charge_tsp_functionally_verified`	Functionally verified? (grey box, establish TSP)
`charge_peripherals_functionality_verified`	Functionally verified? (inside PHM yellow box)
`charge_gnc_peripherals_on`	Are GN&C peripherals on?

Table 1 Events and Decisions Class Methods Overview

Using these methods, we can create the state class (found in Code Block 1), which is the code that defines the behavior of the state. Note that this assumes that the state has already been generated by using the stub script. A lot of code that is generated by the script is removed here for demonstration purposes. Note that the driver code for the state is located in the `run_process` class.

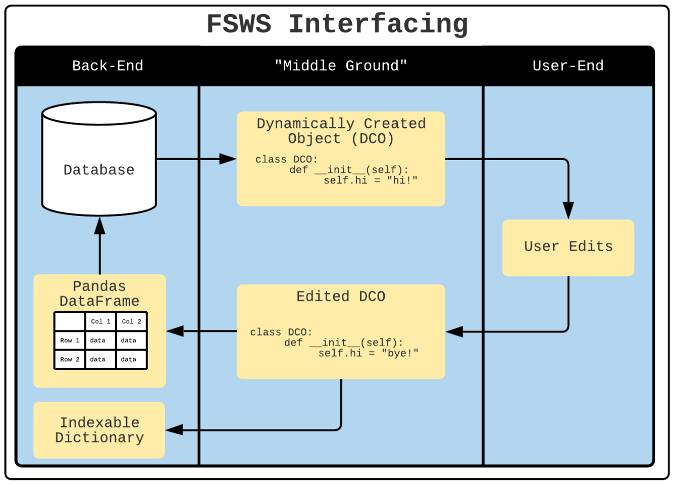
```python
class ChargeState:

    # blah blah blah other code here. Not shown for demo purposes.

    def run_process(self):
        '''''Runs the charge state process.'''

        # print out end/start of new state.
        self.utils.header(self.util_header)

        # ==== YOUR CODE GOES HERE. ====

        self.events.charge_establish_tsp_init_program()
        # Is TSP working properly?
        passed = self.decisions.charge_tsp_functionally_verified()
        if not passed: # it failed
            # Go into safety and check the state stack.
            self.events.add_state_to_state_stack('charge')
            self.events.add_state_to_state_stack('safety1')
            self.events.check_state_stack()
            return # Whenever you have the above line, REMEMBER TO INCLUDE THIS!

        # If it passed, continue this logic (set phm flags)
        self.events.set_phm_flags()
        self.events.read_optimal_charge_vector()

        # Are the peripherals on?
        passed = self.decisions.charge_gnc_peripherals_on()
        if not passed:
            self.events.turn_peripherals_on()
            # Check to make sure they're on.
            passed = self.decisions.charge_peripherals_functionally_verified()
            if not passed:
                # Go into safety 1.
                self.events.add_state_to_state_stack('charge')
                self.events.add_state_to_state_stack('safety1')
                self.events.check_state_stack()
                return

        # All is good, point towards the sun and charge.
        self.events.alg_acquire_sun_position_init()
        self.events.alg_sun_pointing_init()
        self.events.charge()
        self.events.alg_sun_pointing_terminate()

        self.events.check_state_stack()
        return # this return statement is optional since it’s the end of the method.

        # ==============================
```

Code Block 3 Charge State Driver Code

6 Interfacing Between the User and Back Ends – DCOs

This section discusses design choices for communications between the front (user) and back (framework) end of the project. One underlying principle of this project is to remain as close as possible to the built-in python syntax and avoiding 3rd party packages on the user end. After discovering that objects can be dynamically written, the decision to interface with the back end using dynamically created objects was settled upon. Figure 6 shows how a dynamically created object interfaces with the back-end.

The back-end stores data in a pandas DataFrame. This design choice was settled upon because the framework's output was meant to easily be read in with a data analysis library. Thus, it makes sense to use an industry-standard data analysis library to produce an output that is useful (that is, portability purposes).



```
class DCO:
    def __init__(self):
        self.hi = "hi!"
```

```
class DCO:
    def __init__(self):
        self.hi = "bye!"
```

Figure 11 Framework Interfacing

The framework utilizes Dynamically Created Objects, or DCOs, to allow the user to communicate with the back end. Figure 6 explains a general flow of how the back end and the user end communicate through the use of DCO's. These DCO's are generally specific sub-components for a component of the framework (for example, a piece of hardware).

This design choice has many advantages, with the biggest one being the fact that the user will not have to worry about any 3rd party package syntax and functionality. As a byproduct, documentation for the user will be much easier for the developers to convey and for the user understand; consistency is key in a complex framework.

7 Events and Decisions Components

7.1 Events component

The fully stubbed out Events class from Section 5.10 is found below. It inherits the logger, which contains the class for the node to log information in. Note that this class comes pre-built with 3 methods, which are denoted below. Every other method in this class is user generated.

```python
class Events(Logger):

    # === Built-in methods, these you do NOT need to edit. ===

    def add_state_to_state_stack(self, state_str):
        '''Adds a state to the state stack.'''
        # dev check, type checking
        if not isinstance(state_str, str):
            raise ValueError("state_str not string. Found: {}".format(type(state_str)))

        event = Logger.create_new_event() # creates new event node
        event.add_string('Added {} to state queue'.format(state_str))
        Logger.add(event) # adds event to logger.
        Logger.StateQueue.add(state_str)

    def check_state_stack(self):
        ''''Pops the current state off the state stack and transitions into the
        next state.'''

        # built in component
        event = Logger.create_new_event() # creates new event node
        event.add_string('Checking state stack for new state.') # adds a string repr
        Logger.add(event) # adds event to logger.
        Logger.StateStack.check() # State stack functionality

    def set_phm_flags(self):
        ''''Sets the PHM flags for the state it is currently in'''

        event = Logger.create_new_event()
        event.add_string('Setting PHM flags')
        Logger.PHM.set_flags() # PHM functionality
        Logger.add(event)

    # ========================================================

    def read_opt_charge_vector(self):
        event = Logger.create_new_event() # creates new event node
        event.add_string('Reading optimal charge vector') # adds a string representatio
n
        Logger.add(event) # adds it to the logger.

    def charge_establish_tsp_init_program(self):
        event = Logger.create_new_event() # creates new event node
        event.add_string('Initializing TSP') # adds a string representation
        Logger.add(event) # adds it to the logger.

    def turn_peripherals_on(self):
        event = Logger.create_new_event()
        event.add_string("Turning on peripherals")

        # Note: at time of writing, these methods don't exist, but it gets the
        #   point across.
        Logger.turn_on_hardware('rw')
        Logger.turn_on_hardware('imu')
        Logger.turn_on_hardware('st')
        Logger.turn_on_hardware('ss')

        Logger.add(event)

    def alg_acquire_sun_position_init(self):
        event = Logger.create_new_event()
        event.add_string('Acquiring sun position')
        Logger.initiate_algorithm('sun_positioning')
        Logger.add(event)

    def alg_acquire_sun_position_terminate(self):
        event = Logger.create_new_event()
        event.add_string('Terminating sun position algorithm')
        Logger.terminate_algorithm('sun_positioning')
        Logger.add(event)

    def alg_sun_pointing_init(self):
        event = Logger.create_new_event()
        event.add_string('Initiating sun pointing algorithm')
        Logger.terminate_algorithm('sun_pointing')
        Logger.add(event)

    def alg_sun_pointing_terminate(self):
        event = Logger.create_new_event()
        event.add_string('Terminating sun pointing algorithm')
        Logger.terminate_algorithm('sun_pointing')
        Logger.add(event)

    def charge(self):
        event = Logger.create_new_event()
        event.add_string("Charging the satellite's batteries")
        Logger.add(event)
```

Code Block 4 Example Event Class

7.2 Decisions component

This section contains the class that contains the functionality for the decisions. Remember to return the decision that was generated by the decision node, otherwise weird things will happen and it will be very difficult to debug the simulation.

```
class Decisions(Logger):

    def charge_tsp_functionally_verified(self):
        decision = Logger.create_new_decision() # creates new event node
        pass_or_no = decision.decide() # decides whether the diamond passes or fails.

        decision.add_string("TSP Functionally check")
        Logger.add(decision) # adds to the logger.
        return pass_or_no

    def charge_peripherals_functionality_verified(self):
        decision = Logger.create_new_decision()
        pass_or_no = decision.decide() # this is saved inside the decision object and w
ill be auto-written
        decision.add_string("Peripherals init check")
        Logger.add(decision)
        return pass_or_no

    def charge_gnc_peripherals_on(self):
        decision = Logger.create_new_decision()
        pass_or_no = decision.decide()
        decision.add_string("checking if GNC peripherals on.")
        Logger.add(decision)
        return pass_or_no
```

Code Block 5 Example Decision Class

8 Framework Development – Contributions

If a developer wishes to work on this framework, it is suggested understand the following points to have a better grasp on how the framework is connected, and how it interfaces with the user, and what the user should be editing and what the user's coding standards should look like.

There is a semi-hefty learning curve but by following these points, this learning curve will be minimized.

Note that this section does not explain the specifics of each point in this section, but most points are outlined in this document.

Before issuing the first pull request, the developer should understand the following:

1. The purpose of this framework;
2. The architecture of the framework and what components should be edited by the user;
3. The concept of inheritance and what components inherit which components and why;
4. The front and back end data structures;
5. How these data structures are interfaced ("connected") the front and back end;
6. The user-defined coding standard (This is still a work in progress);

Once this is completed, then the GitHub [link] repository can be cloned. Then create a virtual environment and install the dependencies (`pip install requirements.txt` in the main project folder) and then activating the virtual environment: `source <virtual_env_name>/bin/activate`.

9 List of Figures, Tables, and Code Blocks

9.1 Figures

9.2 Tables

9.3 Code Blocks

10 References and Acknowledgements

10.1 References

[1] Li, Haiming, et al. "Research and Improvement of Kruskal Algorithm." *Journal of Computer and Communications*, vol. 05, no. 12, Jan. 2017, pp. 63–69., doi:10.4236/jcc.2017.512007.

[2] Javaid, Muhammad Adeel. "Understanding Dijkstra Algorithm." *SSRN Electronic Journal*, Jan. 2013, doi:10.2139/ssrn.2340905.

[3] McKinney, Wes, and Pandas Development Team. "Pandas: Python Data Analysis Toolkit." *Pandas Documentation*, 1.1.4, 2020, pandas.pydata.org/pandas-docs/stable/index.html#.

[4] "F´ A Flight Software and Embedded Systems Framework." *F´ - Flight Software & Embedded Systems Framework*, NASA, nasa.github.io/fprime/.

10.2 Acknowledgements

I would like to acknowledge the Fall 2020 Flight Software team, consisting of Theo Zinner, Joseph Langan, Grant Robertson, Leah Lee, and Houston Walley for reviewing code and providing advice for the framework and reviewing this documentation on an as-needed basis. I'd like to also acknowledge Michael Halvorson for allowing me to build this simulation to support mission objectives for the NASA CubeSat program.