6-2022

A Model-Based Approach to CubeSat Propulsion and Payload Analysis

Madeline H. Johnson

A MODEL-BASED APPROACH TO CUBESAT PROPULSION AND PAYLOAD ANALYSIS

THESIS

Madeline H. Johnson, Captain, USSF

AFIT-ENV-MS-22-J-063

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENV-MS-22-J-063

A MODEL-BASED APPROACH TO CUBESAT PROPULSION AND PAYLOAD ANALYSIS

Madeline H. Johnson

Captain, USSF

Committee Membership:

Dr. David R. Jacques
Chair

Dr. Richard G. Cobb
Member

Dr. Bradley J. Ayres
Member

AFIT-ENV-MS-22-J-063

Abstract

CubeSats are on the leading edge of low-cost, rapid innovation for the Department of Defense and commercial space industries. Currently, there are only limited ways to increase mission success of CubeSats in terms of component and mission compatibility. The Payload Analysis Tool (PAT), developed by Air Force Institute of Technology (AFIT) students, combines the power of MathWorks' MatLab (Simulink and Stateflow), AGI's Systems Tool Kit (STK), and Dassault System's CATIA Magic System of Systems Architect to analyze payload compatibility on a single CubeSat bus. The PAT simulates a CubeSat mission with a variety of payloads to better understand how the payloads interact with the bus in terms of power, data rate, and memory, but it lacks a propulsion system. This thesis research advances the PAT by including a propulsion system which allows for increased mission time and illustrates the potential power issues CubeSats face during extended mission timelines. The propulsion subsystem was added throughout the Model-Based Systems Engineering (MBSE) framework, including CATIA, STK, and Simulink and all necessary steps are clearly documented in the thesis to aid similar future enhancements. This research specifically looks at AFIT's Grissom-2 mission and builds upon the existing MBSE tools used in the PAT to provide an environment to conduct a propulsion system trade study, as well as further analysis on the CubeSat power over the duration of a propulsion maneuver. Results show that trade studies of various propulsion options can be easily evaluated. This research also displayed how to overcome power generation issues by altering the attitude of the satellite. Potential long-term power issues that may plague the satellite were uncovered and resolved in this research.

Acknowledgements

I would like to thank my husband for his unending support throughout my entire career in the military, to include my time at AFIT. A special acknowledgement to my children who were both born during my time at AFIT. Finally, I would like to thank my advisors and committee, Dr. Jacques, Dr. Cobb, and Dr. Ayres for they continuous guidance and encouragement toward this thesis research.

Madeline H. Johnson

Table of Contents

List of Figures

List of Tables

I. Introduction

1.1 Background and Motivation

Model-Based Systems Engineering (MBSE) is a rapidly growing field, primarily because of the large benefits it promises over document-based approaches [1]. MBSE enables time and cost savings, with early verification and validation of requirements, and increased stakeholder visibility. Specifically, MBSE allows for complex systems to be better understood and developed in order to provide solutions to meet the complexity of Department of Defense (DoD) systems today [2].

Space is one area of development where complex systems are commonplace. Within the DoD, space has emerged as a warfighting domain, requiring the United States (US) to respond with advanced space power "to compete, deter, and win in a complex security environment characterized by great power competition" [3]. In order for the United States to remain competitive in the current and future global, technological, space environment, the DoD must respond with disruptive agility [4]. To support the disruptive agility initiative, the 2020 US National Space Policy discusses how the US must go fast, out-innovating their adversaries, by promoting rapid development and experimentation while still reducing programmatic risk [5]. One way the DoD is pursuing this effort is through the use of MBSE and digital engineering.

MBSE creates a platform for the integrated modeling of a system, removing the need for document-centric systems engineering. MBSE allows for modeling and simulation of space assets which can be accomplished before large amounts of money are invested into a non-proven solution. MBSE is especially beneficial to modular and multi-mission capable systems. CubeSats are one such system. "A CubeSat is a modular, low-cost standardized nanosatellite" introduced

in 1999 [6]. Originally, CubeSats were developed as small, inexpensive satellites "that could be built by students in a relatively short period of time and launched at a low cost" [7]. The CubeSat design standard was based on a cube, with dimensions of 10 x 10 x 10 centimeters, also known as a one unit, or 1U [7]. CubeSats can be configured in multiple ways with 1U increments. Today, CubeSat deployment systems can comprise up to 27U [7]. The International Council on Systems Engineering (INCOSE) has taken interest in MBSE applications to CubeSats, specifically the use of a CubeSat Reference Model [6]. The CubeSat Reference Model provides "an abstract framework for understanding the relationships among the entities of the CubeSat environment" which can then be applicable to multiple mission types due to CubeSat modularity [6]. Modeling the CubeSat design produces a virtual representation of the physical CubeSat, also known as a Digital Twin [8]. Having a Digital Twin of a CubeSat provides many benefits such as an ability to run scenarios and perform risk assessments without requiring and potentially risking the actual satellite, then informing decision making and providing better documentation and communication with stakeholders [8]. Mission scenarios can be prepared before the satellite is physically built to find the best design possible to support the payloads involved in the mission. After satellite launch, the Digital Twin can continue to be used to test software to determine how the satellite will respond to software updates and what effects (intentional and unintentional) the software changes may produce [9].

Each year, the Space Experiments Review Board (SERB) reviews and prioritizes payloads sponsored by DoD agencies based on their mission and need for space flight, to include payloads for AFIT's 6U Grissom bus [10]. Based on the prioritized list, the Space Test Program (STP) then matches the fully integrated Grissom CubeSats with a space launch, with the goal of

launching as many SERB payloads as possible [10]. The Grissom missions will prove the AFIT bus technology, as well as host multiple payload experiments.

To effectively utilize Grissom's payload hosting capabilities, it is necessary to understand the limitations in mission performance when it comes to integrating multiple payloads into one CubeSat bus. As part of the Grissom-2 model, a payload analysis tool was created based on the Grissom-2 mission to analyze the possibilities for hosting multiple payloads [11]. One drawback to the Grissom-2 model is that it does not include a propulsion system which is intended to be a part of the Grissom-2 bus design [11]. By adding a propulsion system, the CubeSat will be able to accomplish a variety of payload missions, to include missions which demand orbital maneuvering and maintenance.

1.2 Problem Statement

Throughout the current literature, there is little to no availability of a tool that can efficiently validate payload mission requirements when considering a CubeSat with multiple payloads and propulsion. There are models of CubeSats, but none that include a fully defined propulsion system. To date, there are only a limited number of CubeSats with propulsion capabilities due to the limited size, weight, and power (SWAP) and few approved CubeSat propulsion systems. This is expected to change as the maturity of miniaturized propulsion progresses.

The usefulness of the propulsion system is dependent on the specific mission. Without a current model of a CubeSat with on-board propulsion, it becomes impossible to validate propulsion performance to support necessary payload mission requirements. The DoD needs a tool that can be used to plug-and-play different payloads and propulsion systems with a CubeSat bus in order to determine which propulsion system is compatible to best meet the specified payload mission requirements.

1.2.1 Research Hypothesis

A parametric propulsion module can be added to AFIT's existing Payload Analysis Tool (PAT) to validate payload mission requirements for a CubeSat with on-board propulsion. This can be accomplished through the use of MBSE, specifically integrating Dassault Systems' CATIA Systems Modeler, MathWorks' MatLab, and AGI's STK in order to effectively model the mission scenario and receive the appropriate results to validate that mission requirements can be met.

1.2.2 Research Objectives & Questions

The main research objective is to effectively add a propulsion module to the existing PAT in order to parametrically validate payload mission requirements. This will involve determining which propulsion system is best suited for the mission based on how well those propulsion systems can meet the specific payload mission requirements. The propulsion systems under investigation include only cold gas thrusters, but have varying elements of interest (fuel type, thrust, size, etc.). When complete, this tool will be useful for the Grissom-2 mission. It will also be modular enough that other members of the CubeSat community could change out CubeSat bus specifications or entire payloads in order to validate that requirements can be met for their specific mission. Using this tool will be especially useful to STP as they assign SERB payloads to various missions throughout the DoD, and it will give greater confidence that the payloads assigned will be compatible with the mission objectives.

The research questions that form this thesis are listed below:

1. How should a propulsion system be modelled and integrated into an MBSE model to validate payload mission success and to what fidelity?

2. What figures of merit should be considered for payload mission analysis when considering CubeSat propulsion with varying payloads?
3. What are the key propulsion performance parameters necessary to evaluate the propulsion system alternatives for a CubeSat mission within a trade study?

1.3 Methodology

The methodology used for this thesis was conducted in the following steps, as specified based on the research questions above:

For Research Question 1:

- Determine what payload mission requirements are mapped to the propulsion system.
- Determine the information/logic/behaviors needed to be captured within a model to integrate a propulsion system.
- Model a propulsion system in CATIA to include propulsion subsystem requirements and attributes.

For Research Question 2:

- Determine the type of mission to include timelines, stakeholders, and the type of propulsion most compatible with the mission.
- Determine the possible metrics necessary for mission analysis.
- Model propulsion metrics in CATIA.
- Simulate the model to see if it meets the specified figures of merit.

For Research Question 3:

- Identify key parameters.

- Using results from the Research Question 2 simulation, identify how the mission could be improved through use of a different propulsion system. Identify where the mission was propulsion limited.
- Simulate model to determine the utility and sufficiency of the propulsion system toward meeting mission requirements.

1.4 Assumptions and Limitations

AFIT's previously developed CubeSat reference architecture and mission modeling tools were intended for application to just CubeSat missions. Therefore, the propulsion system alternatives considered for this research consist of propulsion systems that are applicable and useable for CubeSat missions, in this case cold gas, impulsive, propulsion systems. The propulsion system alternatives may differ in fuel mass, thrusting capabilities, and power consumption but all will be compared based on the same orbit-raising scenario. All of the propulsion systems will also assume a heating time of 1800 seconds, a necessary assumption as there currently is not a detailed thermal model to simulate actual heating demands throughout all phases of the orbit. To modify this model for larger satellites, significant modifications would be necessary. The CubeSat reference architecture as well as STK and MatLab, have their own built-in assumptions. Those assumptions can be found in the "How-To" guides for the CubeSat reference architecture and for the mission modeling tool, as well as noted within the model itself. These guides can be found within the CATIA CubeSat Reference Architecture. Additional assumptions are listed below.

1. CATIA Magic System of Systems Architect and Magic Model Analyst is used for the MBSE portion of work; capabilities and limitations associated with other MBSE tools will vary.

2. STK (Version 12) is the physics engine utilized to generate mission data and for model visualization.
3. MatLab 2021a scripting is used to pass data between CATIA and STK.

1.5 Thesis Overview

The thesis follows a five-chapter format. Chapter I introduces the topic and motivation for the research, describes the problem statement, and presents an overview of the research objectives to include a brief methodology and any assumptions or limitations that may be included in this research. Chapter II discusses the background of the research in greater detail to include previous CubeSat modeling research and a review of propulsion systems currently available. Chapter III describes the methodology used to upgrade the current CubeSat reference architecture and the PAT to include a propulsion system, and to run the proper analysis to validate payload mission requirements. Chapter IV details the results of the updated CubeSat model and PAT. Chapter V summarizes the contributions and limitations of this research and describes areas of future research to further improve the usefulness of the model. Ultimately, the final objective of the thesis effort is to provide CubeSat designers and students with a useful tool to conduct mission payload analysis with a mission that includes propulsion and to document the process throughout the document and detailed appendices (Appendix B-D) to aid in similar enhancements and analyses.

II. Background

2.1 Model-Based Systems Engineering (MBSE)

MBSE applies system models to the entire life-cycle of a system "to support analysis, specification, design, and verification of the system being developed" [12]. Engineering with models has been in place for decades, but with the increasing complexity of systems, there was a dire need for greater consistency between models in order to build the best design to meet the stakeholder's need [13]. One aspect of system complexity is the interconnectivity among systems. "Systems can no longer be treated as stand-alone entities" because they contain aspects of other systems, devices, and humans [12]. As a system progresses, it will change over time as parts of the systems are removed, added, and their overall use adapts [12].

"In MBSE, the "model" is the sole source of truth and reflects the state of system development" which forces a need for continuity in a system when multiple teams are collaborating with multiple models of the system [13]. In order to create greater consistency, specifically with complex systems, MBSE holds all system information in a central repository which "enables the interconnection of model elements, effective information retrieval…reasoning about the system…automatic propagation of design changes, consistency checking, and error identification" [13]. The desired outcome of MBSE is greater design quality, reuse of previous design artifacts, and "improved communications among the development team" [12].

MBSE was first introduced formally by Wayne Wymore in 1993 [12]. At the time, MBSE was mathematically based, expanding as computers and software were advancing [12]. Previous to 1993, many engineering fields such as mechanical, electrical, and software were beginning to use their own computer-based methods by using computer-aided design tools,

moving away from traditional document-based design drawings. The document-based systems engineering approach generates documents and specifications which are then exchanged between all parties involved and a large emphasis is put on controlling the documentation [12]. Due to the number of different documents necessary to accurately articulate the system throughout its lifecycle, a significant amount of resources is poured into ensuring all documentation is consistent, valid, and complete [12]. Making a change to the system becomes difficult to ensure the change is implemented throughout all documentation, or to enable a variant system design. Instead of easily reusing requirements or design work for a system variant, much of the work is reinvented instead of reused, leading to inefficiencies ultimately affecting the cost, schedule, and performance of the program. The document-based approach does use various models and diagramming as part of their documentation; however they are typically for specific parts of the system and do not represent the system as a whole [12]. MBSE allows for emphasis to be placed on controlling the model and design of the system, rather than controlling the documentation about the system [12].

Delligatti splits MBSE into three pillars: modeling language, modeling method, and modeling tool [14]. The modeling language describes the allowed elements and relationships between elements in your model. The modeling language is the foundation on which your model will be communicated [14]. There are multiple modeling languages that can be used, but the Systems Modeling Language (SysML) is the most commonly used throughout MBSE. This thesis will use SysML throughout. SysML uses nine primary diagrams to represent the system. Figure 1 below shows Friedenthal's explanation of the SysML diagram taxonomy [12]:

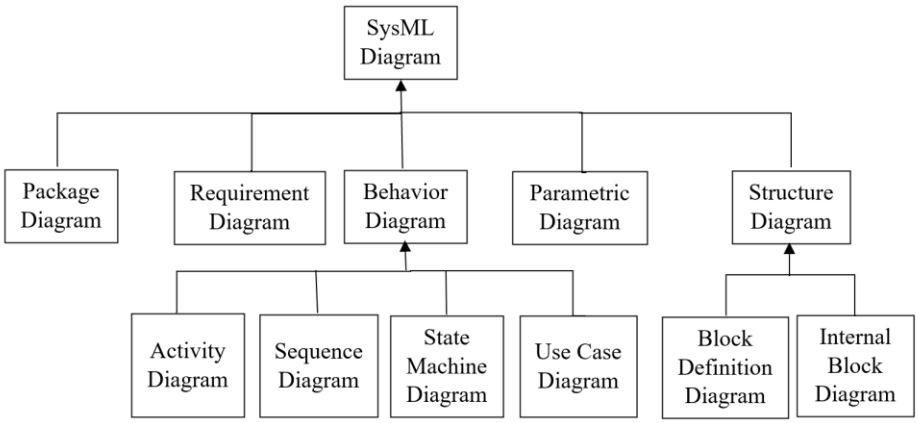

Figure 1 - SysML Diagram Taxonomy [12]

The diagrams primarily used throughout this thesis are Block Definition Diagrams, Parametric Diagrams, and State Machine Diagrams. These three primary diagrams involve the structure of the model, a simulation of the model, and behavior of the model. The simulation for the Grissom-2 mission only captures a portion of the mission duration and therefore the requirements specific to the Grissom-2 mission cannot be validated based on a portion of the mission alone. The primary requirement being looked at in this thesis is if the specified payloads can operate while maintaining sufficient power to avoid entering a fault mode throughout the mission include necessary propulsive maneuvers for orbit maintenance.

Block Definition Diagrams (bdd) are used to define the structural makeup of the system. The primary component of the bdd is a block which describes the system structure [12]. A block is defined by its features, primarily through the use of properties and parts. Properties are the defining characteristics of the block, while parts are what the block is made of [12]. Depending on the system and defined methodology, parts may be represented as other blocks and threaded together through a hierarchy, or it can be defined within the individual block [12].

Parametric Diagrams use systems of equations to constrain the properties of blocks, and are ultimately used for analyzing the system. Parametric diagrams can evaluate multiple aspects of the system to include performance, reliability, and cost which in turn can be used in a trade-study to discover the best alternative based on the provided constraints [12]. Constraint blocks are used to define mathematical equations "with the parameters of the equations being bound to the properties of the system being analyzed" as well as constraint properties [12]. Constraint parameters can be associated with properties of the blocks, and the constraint equation defines the dependencies between the parameters [12]. Binding connectors are used within the parametric diagram to "express equality relationships between their two ends". This is then used to connect multiple equations and create complex sets of equations [12]. The constraint block ultimately outputs constraint parameters of interest to the user based on the mathematical model used. Parametric diagrams can be integrated with non-SysML tools like MatLab, increasing the functionality and power of analysis [12].

State Machine Diagrams model the "state-dependent behavior of a block throughout its lifecycle" defined by different states and transitions between the states [12]. Transitions are defined by triggers, guards, and effects, ultimately showing what needs to happen for something to change states [12]. The state may contain entry, do, and exit behaviors, explaining what happens to enter and exit the state, as well as what behavior occurs within the state.

The modeling method is "a documented set of design tasks that ensures that everyone on the team is building the system model consistently and working toward a common end point" [14]. Discovering the modeling method begins with defining the purpose of the model, followed by the scope. The scope will ultimately determine your modeling method. The modeling method can be tailored using existing methods, or the modeling team can create a new method specific to

their purpose and scope. Estefan describes methodology “as the collection of related processes, methods, and tools used to support the discipline of systems engineering in a “model-based” or “model-driven” context” [15]. Table 1 summarized by Brown contains methodologies typically used today [16]:

Table 1 - MBSE Methodologies [16]

Method	**Description**
Functional Analysis and Allocation	Decomposes functions of the system and then allocates the functions to modeled components
Object-Oriented Systems Engineering Method	Leverages object-oriented concepts for a top-down, scenario driven, and flexible system development.
Control and State Based	Developed and implemented by the Jet Propulsion Laboratory at the National Aeronautics and Space Administration (NASA) and focuses on modeling states and controls of evolving systems

Modeling tools “are designed and implemented to comply with the rules of one or more modeling languages, enabling you to construct well-formed models in those languages” [14]. Modeling tools are different from diagramming tools because in diagramming tools there is no underlying model ensuring consistency. Many diagramming tools are used in document-based systems engineering. The modeling tools allow one to construct a model defining the elements and relationships while using diagrams to display different views of the model [14]. There are many modeling tools available that typically contain various modeling languages [14]. A few examples of tools found in industry include Enterprise Architect, CATIA, MagicDraw, Rational

Rhapsody, and Visual Paradigm. There are also open-sourced tools such as Modelio, SysML Architect, and Papyrus SysML [17] .

2.2 CubeSats

CubeSats were first developed by students at California Polytechnic State University (Cal Poly) in partnership with Stanford's Space Systems Development Laboratory (SSDL) in an effort to "develop a picosatellite standard that significantly reduces the cost and development time of student satellites" [18]. With a shorter development time, students were able to participate in the entire spacecraft development process, beginning with requirements development and ending with satellite operations, throughout their time at the university [19]. CubeSats also were developed to provide a platform for in-space experimentation [18]. CubeSats began as a 10 cm cube (10x10x10 cm also known as 1U), weighing less than one kilogram. In tandem with the development of the CubeSat, Cal Poly was also developing a CubeSat deployer that would contain the CubeSat throughout the launch, protecting the launch vehicle, as well as providing a viable interface compatible with multiple launch vehicles [18]. The first deployer was called the Poly Picosatellite Orbital Deployer (P-POD) and could house up to three 1U CubeSats.

The first CubeSat was launched in 2003and because of their small size and contained P-POD environment, CubeSats began to become ideal for rideshare opportunities because there was little fear of damaging the launch vehicle or primary satellite [20]. Soon after, CubeSats became a world-wide phenomenon. Commercial partners who had once never dreamed of experimenting in space due to the extreme costs and time commitment, were now able to access the space sector [20]. Due to the success of the 1U CubeSats, iterations were made to create a standard for larger CubeSats consisting of 6U, 12U and 27U [21].

This thesis includes a mission scenario concerning a 6U CubeSat and will follow the design specifications as provided by the CubeSat Program, Cal Poly San Luis Obispo (SLO). "The CubeSat standard specification document gives a clear description of the constrains that the satellite should address in order to be considered a CubeSat" [20]. Even with the specifications provided by Cal Poly, the overall design of the CubeSat is largely affected by the mission and system requirements, to include the requirements of the launch provider [22]. In order to interface with the launch vehicle, the CubeSat is temporarily housed in a CubeSat dispenser for the launch of the satellite. The dispenser largely influences the design of the CubeSat as it affects the size, electrical connections, and how the satellite is ejected. The Grissom missions use the 6U Planetary Systems Corporation (PSC) canisterized satellite dispenser (CSD) and are therefore built according to the PSC CSD payload specification guidelines [23]. The launch vehicle sends a signal to the dispenser when the CubeSat is ready to be ejected into its proper orbit [22]. The 6U CubeSat Design Specification includes many requirements that involve how to keep the launch vehicle and other rideshare satellites safe throughout the launch, to include types of materials that cannot be used, various mechanical, electrical, operational, and testing requirements [22]. A few requirements of particular importance to propulsion systems on a CubeSat include sections 3.1.3, which states "Any propulsion systems shall be designed, integrated, and tested in accordance with AFSPCMAN 91-710 Volume 3" and 3.1.4 which states "Propulsion systems shall have at least 3 independent inhibits to activation" [22]. Many CubeSat propulsion systems available today meet these requirements, but it is something to keep in mind when choosing which propulsion system is most appropriate for the mission.

Outside of the standard CubeSat requirements, much of how the CubeSat is designed is determined by the mission payload and the bus. The primary mission of the satellite is derived

from the satellite's payload. The CubeSat must support the payload through all mission phases, to include any loads, operating conditions, and mission configuration that the payload might need to attain [20]. The CubeSat bus contains everything else in the satellite, besides the payload. "Bus elements that need to be considered in the structural design include the Electronic Power System (EPS), the Telecommunications, Tracking, and Command (TT&C) System, the Attitude Determination and Control System (ADCS), the Orbital Determination and Control System (ODCS)," the electronics in general to include the Command and Data Handling System (C&DH), and the thermal management system [20]. If a CubeSat needs propulsion, the propulsion system is included as part of the CubeSat bus. A summary of the CubeSat bus subsystems is found in Table 2, as provided by Brown [16]:

Table 2 - CubeSat Bus Subsystems [16]

Subsystem	**Function**
C&DH	This is the brain of the CubeSat. It is the processor and electronics necessary to receive and distribute commands, and store/forward all data.
EPS	This subsystem typically contains all of the components necessary for power generation, distribution, and storage (batteries, solar arrays wiring, etc.).
ADCS	This subsystem is responsible for controlling the CubeSat attitude and pointing. This typically requires sensor, actuators of some kind, and software.
ODCS	This subsystem is responsible for achieving and maintaining the specified CubeSat orbit, to include maneuvering and navigating.
TT&C	This subsystem is responsible for the CubeSat communications. It contains all the radios and antennas to communicate with the ground and other spacecraft.
Propulsion	While not all CubeSats have propulsion, this subsystem is responsible for providing the CubeSat's thrust (delta V) requirements and fuel storage.
Note: Structures and Thermal are not discussed here as subsystems.	

2.3 Propulsion Systems

Propulsion has been used since the beginning of the space-age in order to launch spacecraft into space, and subsequently to maneuver the spacecraft after it has been separated from the launch vehicle [24]. “A propulsion system is the primary mobility system of a spacecraft and helps with various maneuvering operations like orbit changing and station keeping” [25]. For small spacecraft, there are typically three categories that differentiate between types of propulsion to include chemical, electrical, and propellant-less thrusters [24].

Chemical propulsion has been used since the beginning of propulsion technology and therefore has greater flight heritage and capability, remaining the “in-space propulsion technology of choice when their total impulse capability is sufficient to meet mission requirements” [24]. Chemical thrusters are used primarily “when high thrust or rapid maneuvers are required” [24]. Chemical propulsion systems typically “require on-board power only to regulate (initiate and terminate) the propulsion process,” but power is not actively required for operation [25]. Included in chemical propulsion are: “hydrazine-based systems, other mono- or bipropellant systems, hybrids, cold/warm gas systems, and solid propellants” [24]. Out of all of the chemical propulsion systems listed above, cold gas systems are the only ones that have been proven on a CubeSat and therefore will be discussed in further detail.

“Cold gas systems are relatively simple systems that provide limited spacecraft propulsion and are one of the most mature technologies for small spacecraft” [24]. They also are well-suited for small busses because they have low cost and low complexity [24]. Cold gas systems thrust by the “controlled ejection of compressed liquid or gaseous propellant” [25] and “the thrust produced is directly proportional to the pressure of the propellant inside the tank” [25]. “Cold gas propulsion technology relies on gas expansion through a nozzle to generate

thrust. The propellant is allowed to flow from a propellant storage tank to a converging/diverging nozzle out of which it expands to open space" [26]. The thrust generated from a cold gas thruster can be between tens of millinewtons to tens of newtons [26]. Cold gas propellant is typically inert and non-toxic which make them suitable for rideshare missions due to meeting many of the "do no harm" requirements of the primary payload [24]. Cold gas thrusters use propellants such as He, N_2, and Freon-14. H_2 can be used but may have leaking issues due to the small size of the molecules [27]. Cold gas propellant also has a smaller specific impulse (a measure of thrust efficiency) when compared to other chemical or electric propulsion systems, ranging between ten to a few hundred seconds [26]. Therefore, they are typically "used in cases when the thrust and specific impulse (Isp) requirements are low and a small impulse is important" [27]. The total impulse suitable for a cold gas thruster tops out at 22,000 N-s, making them most suitable for attitude control and small delta V changes [27]. Due to cold gas propellant's suitability in rideshare missions, they have a well-documented flight heritage with CubeSats.

Electric propulsion systems "actively require on-board power for their operation" [25] and typically a propellant is "accelerated through the conversion of electrical energy into kinetic energy" [24]. Electric propulsion can provide a greater total impulse than chemical systems, but the research required to develop electric propulsion systems comes at a steep cost compared to chemical systems [24]. Electric propulsion systems used today on small spacecraft include resistojet, arcjet, electrodeless thrusters, electrospray, gridded-ion, Hall-effect, pulsed plasma, vacuum arc, and ambipolar. Though many of these propulsion systems are used on small spacecraft, the electrospray thruster is the only one proven on a CubeSat.

Propellant-less thrusters include solar sails and electrodynamic tethers. Solar sails use "photons emitted by the sun to provide a propulsive force for the spacecraft" [28]. There have

been a few CubeSats that have utilized solar sails, but typically due to a high-drag, low-Earth orbit, have unsuccessfully generated any usable thrust [28]. Electrodynamic tethers consist of an electrically conductive wire with current flow that when interacting with the ambient magnetic field around a planet, produces a Lorentz force which can be used for orbit raising and lowering [24]. “Tethers cannot be used for attitude control or relative motion between satellites” and therefore are primarily used for de-orbiting [28] [24]. Few CubeSats have successfully used tethers as there is inconclusive data if the tethers ever deployed as intended [28].

In order to determine which propulsion system best fits the satellite, the mission and objectives must be clearly set out. The following steps are used to select a propulsion system necessary for a mission:

1. Determine all the functions the propulsion system must accomplish;
2. Determine the required delta V and thrust for orbit insertion and orbit maintenance;
3. Determine the required total impulse, thrust level for control authority, and duty cycles for attitude control;
4. List propulsion system options;
5. Estimate key parameters for each option,
6. Conduct trade studies [27].

The first step is to understand the objectives of the mission, which includes determining types of maneuvers that are required [27]. Typically, CubeSats may perform the following maneuvers: “midcourse corrections, orbit insertions, station keeping and pointing once the mission orbit is achieved, and, if necessary, disposal.” [20]. The payload and desired lifetime of the satellite largely influences which types of maneuvers will be necessary throughout its mission. The mission objectives should include which orbit the satellite will be launched into,

any satellite constraints, and what level of cost, schedule, performance, and risk the mission is able to accept [27].

Steps 2 and 3 require quantitative analysis on performance requirements to include "thruster efficiency, specific impulse (Isp), total impulse, and impulse density" [28]. Thruster efficiency describes how well the system converts power into thrust and is often arbitrary depending on the type of propulsion system being used; therefore, it is not the preferred method of comparing CubeSat propulsion systems [28]. Specific impulse "is the measure of the total impulse delivered per unit of propellant consumed" [28]. Specific impulse can be compared across propulsion types, but it is not useful for comparing total impulse or time required for thrust [28]. "Total impulse will give an indication of the amount of momentum change that a propulsion system can provide" which is highly influenced by the amount of propellant the propulsion system can carry; therefore, it may be variable when comparing between systems [28]. To combat the discrepancies between propellant amount, one should assume the same amount of propellant is used for each system when conducting a propulsion system comparison [28]. At times, this is inconsistent due to the mass and volume constraints of the CubeSat. To account for these differences, one can use impulse density which is "a measure of the total impulse per unit volume of the propellant" [28]. By using the above performance parameters, one can make a better comparison between propulsion systems and their respective usefulness for the mission, than using one factor alone [28].

Another performance requirement that needs to be considered and calculated for comparison is delta V [27]. Thrusting maneuvers are split into two main categories, high-thrust or impulsive maneuvers, and low-thrust or continuous maneuvers. Chemical propulsion systems are typically "designed to satisfy high-thrust impulsive maneuvers", while electric propulsion

systems typically perform low-thrust maneuvers [24]. Chemical thrusters offer lower specific impulse than electrical thrusters, but have a higher thrust to power ratio. Figure 2 shows NASA's view on differentiating between thruster types:

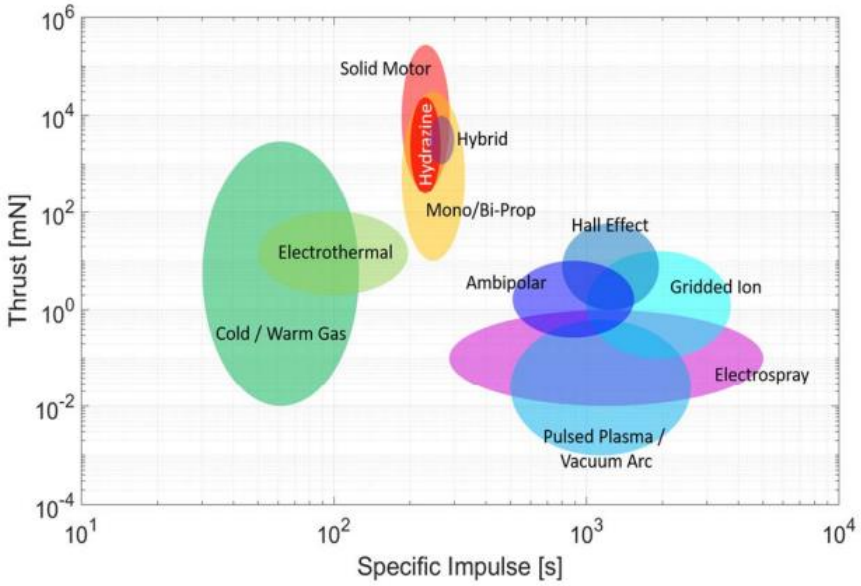

Figure 2 -Thruster Types based on Thrust and Isp [24]

Chemical systems can use the ideal rocket equation, seen below, for calculating delta V [25].

$$v_e = g_o I_{sp}$$

$$\Delta v = v_e \ln\left(\frac{m_i}{m_f}\right)$$

The ideal rocket equation is based on the equivalent engine exhaust velocity (v_e) which is calculated based on the gravitational force and the Isp of the propellant, as well as the initial mass of the object (m_i), the final mass of the object after burnout time (m_f). This can be calculated by subtracting the propellant mass from the initial mass [27]. Electric systems typically operate continuously, which means that their "maneuver accelerations are integrated over the total burn duration" resulting in misleading delta V calculations [20]. In this case, using

the other figures of merit mentioned above may produce a better comparison than delta V when comparing chemical and electric thruster systems [20]. Table 3 from the CubeSat Handbook breaking down specific figures of merit used to determine the performance parameters listed above [20]:

Table 3 – CubeSat Handbook Propulsion Figures of Merit [20]

Figures of Merit	**Units (SI)**	**Definition**
Thrust (F)	N	Total amount of force produced by a system or thruster.
Specific impulse (Isp)	s	Measures propellant performance by quantifying the total impulse per unit mass of propellant.
System change in velocity	m/s	Quantifies system ability to change its velocity based on propellant performance and spacecraft mass.
Density specific impulse (I_d)	kg-s/L	Used to compare propellant performance for given Isp and density. This is generally how well the propellant packages.
Total impulse (I_t)	Ns	Change in momentum given by integrating thrust over a given burn time. Quantifies total amount of force produced by the propellant.
Volumetric impulse	Ns/L or Ns/U	This efficiency parameter used by SmallSat propulsion systems describes the amount of total impulse (Ns) a system imparts to a body per unit volume (U or L).
Propellant mass fraction	None	Quantifies the efficiency of a propulsion system to move a given mass (m_f).

Step 4 consists of listing the available propulsion systems based on the requirements found in steps 1-3. Cold gas thrusters are the primary thruster considered in this thesis as they have shown mission success on previous CubeSat missions. The specific analysis of requirements and scenario determination will be found in Chapters III and IV.

Step 5 takes into account each propulsion system being analyzed and understanding the parameters, as stated for comparison in steps 2 and 3 above. Other than the actual thrusting parameters such as specific impulse and thrust, other parameters such as total mass, power requirements, system volume constraints, a number of “ilities” (reliability, manufacturability, storability, scalability, vulnerability, etc.), cost, schedule, risk, and toxicity of propellant are all

considered [27]. Typically with propulsion systems, volume is a limiting requirement before mass, therefore "volumetric efficiency of a propulsion system is crucial" [20]. Throughout the mission, the mass of the CubeSat will vary due to propellant use, so determining the total mass throughout the mission may be a factor in how much propellant is actually necessary [20]. Available power is a main differentiator between chemical and electric propulsion systems. Electric systems "actively require on-board power for their operation" [25]. Because of this, electric propulsion systems typically require larger solar arrays and batteries in order to produce enough power to propel the satellite, which inherently affects the other subsystems and the overall power and mass budget [20]. Considering the "ilities", if a system has been successfully used on a CubeSat mission in the past and can meet the requirements, there is a greater likelihood that the mission will be successful when compared to use of a non-space proven system, and therefore may be a better choice for a thruster [27].

Step 6 concludes with choosing the actual propulsion system that will best meet the requirements of the mission through a trade study. Each quantitative requirement will be weighted and each propulsion system will be ranked to make the ultimate decision. This process is iterative, and just because one propulsion system looks like the best option up front, further study may cause the team to make needed adjustments throughout design and test [27].

The goal of this research is to simplify the iteration process by incorporating the method just discussed into the MBSE framework used at AFIT.

2.4 AFIT Grissom Project

The Grissom project is a set of two separate 6U CubeSat missions titled Grissom-1, and Grissom-2, to be built and deployed in that order. The overall mission of Grissom is for AFIT to enable rapid CubeSat payload integration for the Space Force and DoD missions [11]. Grissom is

a collaboration across many government agencies with the goal of producing a government-off-the-shelf (GOTS) platform for hosting future experiments. Government payloads should easily integrate into and be tested in CubeSats, but custom CubeSats require long development timelines and commercial-off-the-shelf (COTS) solutions require yearly contracts [11]. A successful Grissom mission would prove the capabilities of a government-owned hardware, software, and development team [11]. AFIT's goal is to launch one CubeSat per year which would allow AFIT students the additional benefit of a hands-on CubeSat build experience, preparing them for future work relevant to their follow-on assignment.

Grissom-1 is a 6U CubeSat with the objective of space qualifying the AFIT designed and built Grissom 6U CubeSat bus. A second objective is to validate the software architecture with incremental on-orbit software updates. Grissom-1 has two payloads, Naval Information Warfare Center's (NIWC) Nano-Satellite Tracking Experiment (NTE), a passive RADAR retro-reflector, and Los Alamos National Lab's (LANL) Extremely Low Resource Optical Identifier (ELROI), a spacecraft identification beacon [11]. Grissom-2 will build off of lessons learned from Grissom-1 to demonstrate rapid payload integration. A successful Grissom-2 mission will provide further Grissom bus flight heritage and prove the modular architecture capabilities for payload integration. A list of possible secondary payloads for the Grissom-2 mission are listed in Table 4 from Brown [16]:

Table 4 - Grissom-2 Secondary Payloads [16]

Payload	Description
TeraHertz Imaging Camera	Enables remote satellite servicing and situational awareness applications (1THz-10THz)
Software Defined Radio for MC3 Validation	Provides in-space calibration source for validating the MC3 network S-band and X-band and provides a customizable downlink capability for communications demonstration with the MC3 network.
AFIT/NIWC Flight Radio	GOTS solution candidate to replace the Cadet Radio.
Satellite Path-Agnostic Communications Experiment (SPACE)	Payload sends mission data to ground by utilizing simultaneous transmission methods, to include commercial networks.
Busek Green Thruster	Stable 'green' propellant thruster that delivers 500 mN of thrust at 220-225 seconds specific impulse.

The design specifics for the Grissom-2 mission are what make the CubeSat model in this research unique, and the compatibility of the design decisions ultimately determine the outcome of mission success. Below is an explanation of previous work within and outside of AFIT which contributed to AFIT's current CubeSat MBSE modeling and analysis.

2.5 Previous Work

2.5.1 Previous Industry Work: MBSE Applied to Radio Aurora Explore CubeSat

As a part of the INCOSE's MBSE Initiative Systems Engineering Vision 2020, and in combination with the Space Systems Challenge Team, the INCOSE Space Systems Working Group (SSWG) began work to "demonstrate the application of MBSE to a realistic mission in the space systems domain" [29]. The project was centered around the Radio Aurora Explorer (RAX) CubeSat, which "periodically passes over high-powered, ground-based radar stations" collecting data and transmitting that data by radar [29]. The purpose of this project was to model RAX in order to "prove out the applicability of MBSE for modeling operational space missions" [29]. The accurate modeling of the actual satellite was not the focused intention [29]. The RAX

mission was defined through SysML, with detailed modeling of payload energy and data collection and management [29]. Block definition diagrams, parametric diagrams, activity diagrams, and state machines were used to diagram various aspects of the mission. STK and MatLab were used to define the analytical models, passing information to the parametric diagrams, which was then analyzed and passed to the behavioral diagrams. A visualization of the various tools used is seen in Figure 3 below [29]:

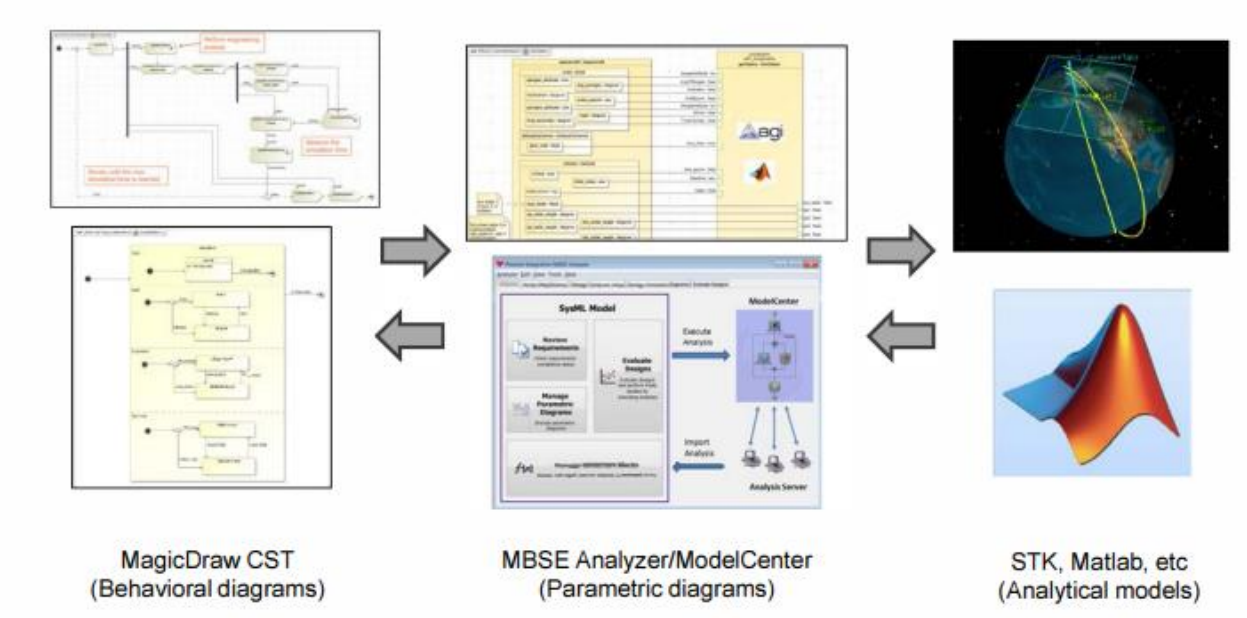

Figure 3 - RAX Modeling Tools Flow [29]

Throughout the mission simulation, time history is kept in order to track the energy states and data download states. The information collected "was used to perform trade studies of key design parameters of components and mission" [29]. Components under consideration were the solar panels and the battery, while mission variations were considered for orbit and ground station selection. The RAX dynamic model was found to be successful, performing mission analysis and trade studies to evaluate different design parameters and mission performance. The

process and data collection from this study can be used for future CubeSat mission teams to develop their own model and perform trade studies using MBSE [29].

2.5.2 Previous AFIT Work: AFIT Mission Modeling Tool

Over the years numerous AFIT students have worked to combine rapid analytical analyses into automated tools for mission design. The following provides a brief chronology of the development towards an MBSE solution.

The Mission Modeling Tool (MMT) was first developed by Mr. Judson McCarty in 2010 as a satellite mission analysis tool called Satellite Simulator (SatSim) using Simulink [30]. SatSim was then further developed by Captain Blythe Andrews in 2012 as the Colony II 3U Bus Mission Modeling Tool (C2BMMT). The C2BMMT utilizes STK, Excel, MatLab, and Simulink to analyze multiple satellite subsystems at the same time to validate the mission concept of operations for the Colony II Bus [31]. A visual representation of the tools used in the C2BMMT Framework is seen below [31]:

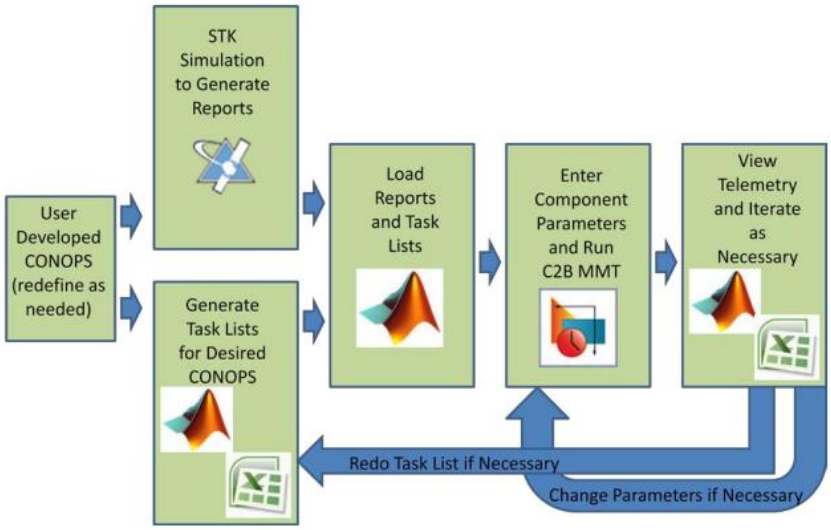

Figure 4 - C2BMMT Framework Tools [31]

Capt Angela Hatch further modified C2BMMT into the MMT which conducted detailed mission planning for the AFIT Bus, to include a Jet Propulsion Lab (JPL) Microfluidic Electrospray Propulsion (MEP) thruster system and an improved run time [32]. Hatch's MMT analyzes a variety of orbits, power consumption, and delta V maneuvers using the MEP thrusters [32]. STK's Astrogator was used to model the thruster maneuvers, instead of calculating the data via a Simulink model as seen in SatSim's model. The MMT m-files were then modified to process the thruster data produced by STK [32]. Below is the MMT user process as developed by Hatch [32]:

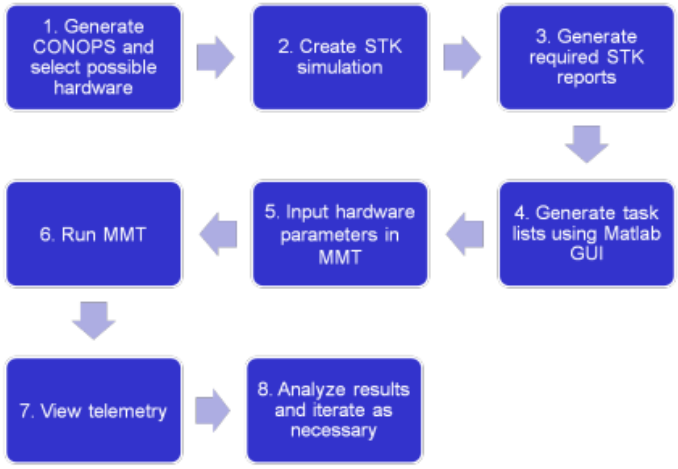

Figure 5 - MMT User Process [32]

Following Hatch's work, 2d Lt Heather Udell further modified the MMT to validate the Low-Earth Orbiting Tele-Imaging Satellite (LOTIS) power system and memory capacity [33]. Udell removed Hatch's thruster from the MMT in order for the model to be used by a CubeSat without a thruster and appropriately model the 6U LOTIS mission [33]. The MMT was updated to then current versions of STK and MatLab, and day-in-the-life LOTIS constellation and design analysis could be accomplished. The MMT was limited to short scenarios due to the number of satellites and amount of data produced during the simulation [33].

Concurrent with Udell's work, Capt Benjamin Jewell began work on the CubeSat State Analysis Tool (CSAT). The CSAT applies MBSE state analysis to the conceptual development phase of CubeSat missions [34]. CSAT utilizes Stateflow to model the states, modes, and logic necessary for CubeSat modeling while providing a tool to "analyze CubeSat use cases and state machines using an executable state chart" [34]. The CSAT focuses on the functional design aspects of the CubeSat mission, rather than the physical component performance [34].

In an effort to combine the efforts of Udell and Jewell, Capt Joshua Loudermilk (reference Figure 6) created the Logic-based MMT (LMMT) "capable of triggering fault modes and employing corrective measures during simulations" using Stateflow [35]. These additions allow for the user to validate mission requirements throughout the simulation, ultimately improving the mission assurance for future CubeSat operations [35].

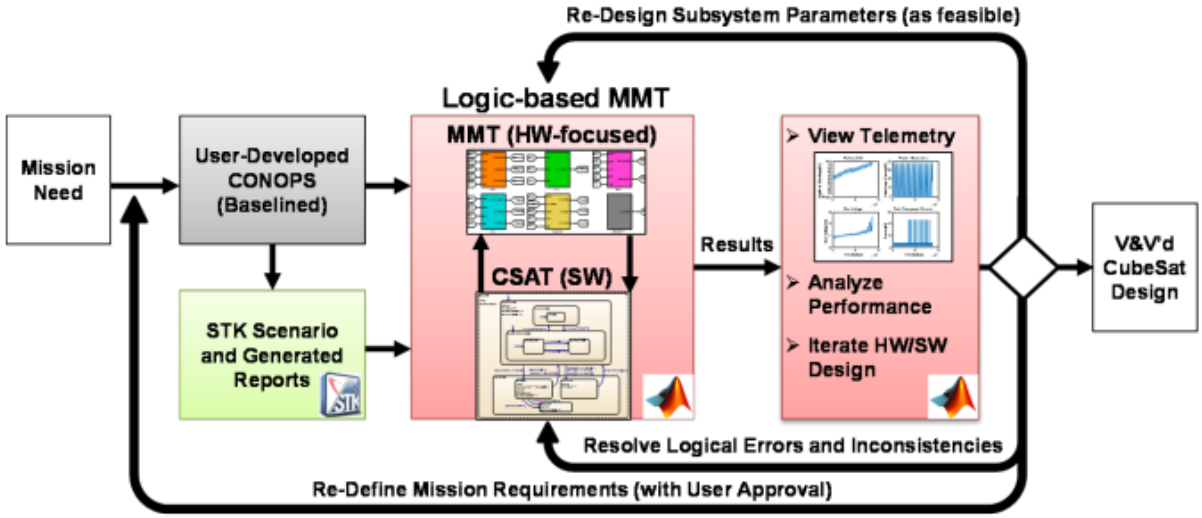

Figure 6 – Loudermilk's Development Process [35]

Capt Justin Sadowski took Loudermilk's LMMT and looked into comparing the static modeling of the current LMMT to the usefulness of dynamic modeling [36]. Throughout the scenario, the environmental model may change in such a way as to make the current satellite's

mode of operation obsolete, thereby necessitating the model to be dynamically updated with changes and repropagating the environmental model [36]. Sadowski analyzed multiple use cases to determine when a dynamic model would produce more accurate results than a static model through the use of his DyLoMMT model. DyLoMMT still utilizes STK, MatLab, and Simulink/Stateflow, but repropagates the scenario if necessary based upon the state diagram logic found in Stateflow [36].

In 2021, Capt Kyla Brown further expanded the LMMT in its static state to reflect AFIT's Grissom Bus, including multiple payloads, and introducing Cameo through Parametric Diagrams and MatLab scripting [16]. Brown's new tool is called the Payload Analysis Tool (PAT) and the process overview is shown below in Figure 7:

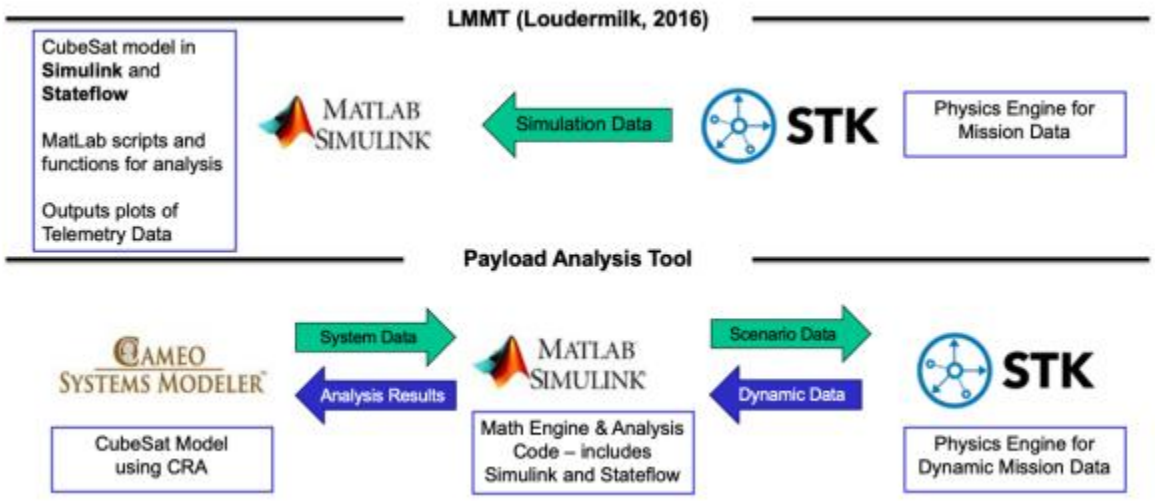

Figure 7 - LMMT vs PAT Process [16]

The Grissom mission's design, requirements, and behaviors were captured in Cameo. This allowed for multiple mission scenarios to be analyzed by easily adjusting parameters through the parametric diagram and saving the analysis results as different instances in Cameo [16]. Brown's updates to the LMMT led to greater model fidelity within Simulink and STK,

specifically relating to the battery logic and addition of tools to analyze up to four payloads on a single CubeSat bus [16]. The MatLab code used to run the scenario was refined to integrate all programs with one user interface, analyzing power, data rates, and fault modes all in one place [16].

The PAT is an integrated model that utilizes Cameo Systems Modeler to drive the tool, and passes information through MatLab to the physics based model in STK, to ultimately pass back through MatLab for analysis, and finalize results within Cameo. The user inserts parameters into the parametric diagram within Cameo, and then runs the entire scenario from Cameo. The specific satellite mission scenario is scripted in MatLab. The parameters necessary for the scenario are called from Cameo, but the actual scenario dynamics are preprogrammed into MatLab and cannot be altered without editing the script directly. Brown's model is a static evaluation of the Grissom Bus's ability to accomplish the mission scenario, as the scenario is run in its entirety in STK followed by analysis of the scenario in Simulink. There is no feedback loop to model dynamic behavior of the model during the scenario. Below is a visualization of the different modeling tools integrated within the PAT:

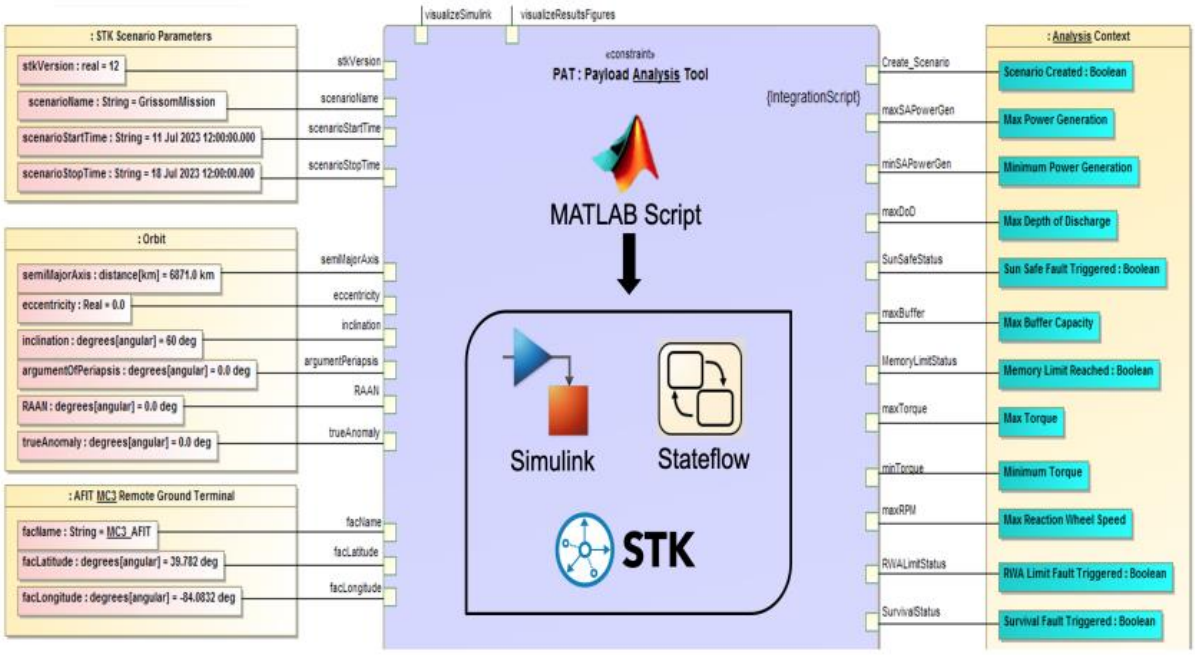

Figure 8 - PAT Organization [16]

The PAT is split into three different tools. The first tool of interest is Cameo Systems Modeler. Cameo holds the satellite model, to include all parameters necessary for the analysis. The Cameo model is based upon the CubeSat Reference Architecture (CRA) created by Capt Sean Kelly, which is further explained below [37]. The CRA is split into four main packages: 1-Guidance, 2-CubeSat Component Library, 3-Generic CubeSat Model, and 4-CubeSat Physical Models. Within the fourth package, CubeSat Physical Models, is the Grissom-2 Model package. The Grissom-2 Model contains all necessary data for the PAT to run, primarily located within package 4-Analysis. Below is a breakdown of the Cameo containment tree for the CRA and PAT:

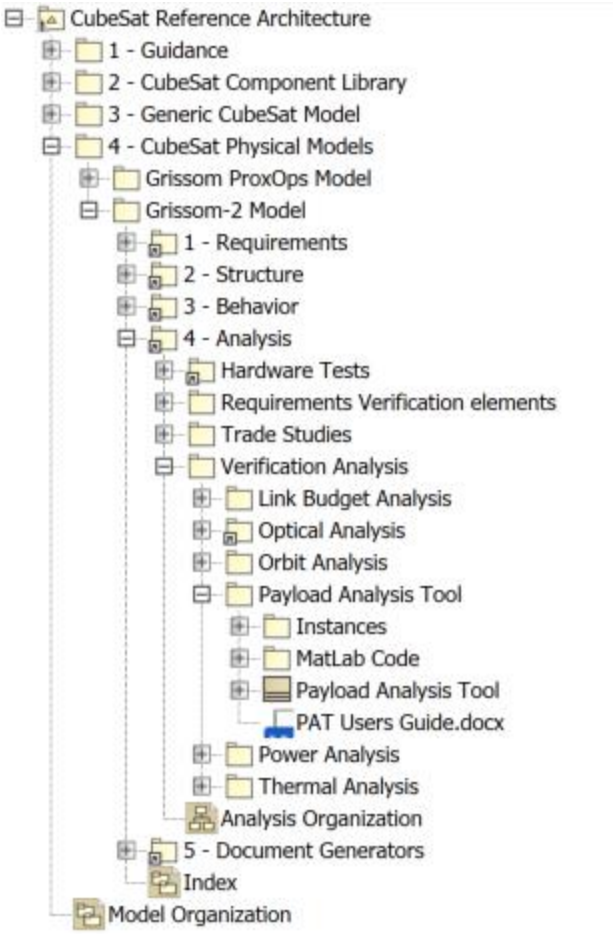

Figure 9 - Cameo Containment Tree for the CRA and PAT [16]

The Payload Analysis Tool block contains the Payload Analysis Tool parametric diagram which is where the PAT is run from, and where all parameters are housed. The value properties located on the left side of the parametric diagram are necessary for input into the MatLab script stereotyped as a constraint parameter and located in a port connected to the main PAT constraint block. All parameters are specific value properties from a block located in package 2-Structure within the Grissom-2 Model. The value properties for the following blocks can be changed, therefore changing the STK scenario within the bounds of the MatLab scripting: Simulation Options, STK Scenario Parameters, Orbit, AFIT MC3 Remote Ground Terminal, CubeSat, Target1, Target2. The blocks located within the CubeSat Bus portion of the parametric diagram

hold parameters necessary for the Simulink/Stateflow models and also can be changed as different subsystem components are chosen for the mission. The PAT holds three locations for payload specific information, with ports available to add a fourth. Having all parameters easily accessible makes conducting trade studies and mission analysis much more effective.

On the left side of the parametric diagram in Figure 8 is the Analysis Context. The Analysis Context holds the discrete value properties produced by running the tool, which then can be traced back to the requirements. These value properties can be expressed in an instance table, useful for further scenario analysis.

In the middle of the parametric diagram is the large constraint block which executes the *IntegrationScript* and *makefigures* script to automate the mission scenario creation in STK, and the PAT in Simulink/Stateflow. The brains behind the PAT lives within the MatLab *IntegrationScript*. The *IntegrationScript* is called from Cameo to begin the simulation. The script is able to create the STK scenario, generate reports, parse data, and run the PAT in Simulink/Stateflow. The script outputs a *makefigures* MatLab file and inserts back into Cameo the minimum and maximum discrete values, as seen in the Analysis Context block. Though the parameters and variables necessary for the script are pulled from Cameo, the STK scenario is first programmed within the *IntegrationScript* MatLab file to establish the STK scenario including the mission objectives.

The STK scenario considered in Brown's work consisted of a satellite orbiting the Earth for one week, as specified by the scenarioStartTime and scenarioStopTime value properties of the STK Scenario Parameters block in Cameo. There is one ground station, at AFIT, and two targets positioned at different points throughout the globe. Throughout the scenario, the satellite orbits the Earth and comes in view of the various targets and ground stations. The access data

between the satellite and ground station and targets is used in the analysis of the payloads. Due to the scenario only lasting one week, there is no need to account for any perturbations of the orbit.

The PAT Simulink/Stateflow model is also called from the *IntegrationScript*. Within Simulink, there are three main models: External Models, CubeSat Model, and Telemetry Storage. External Models utilizes STK parameters and executes the CubeSat State Analysis Tool (CSAT). CubeSat Model consists of the CubeSat subsystem Simulink Diagrams, and Telemetry Storage is used for analysis plots and uses the *makefigures* command after the simulation is complete.

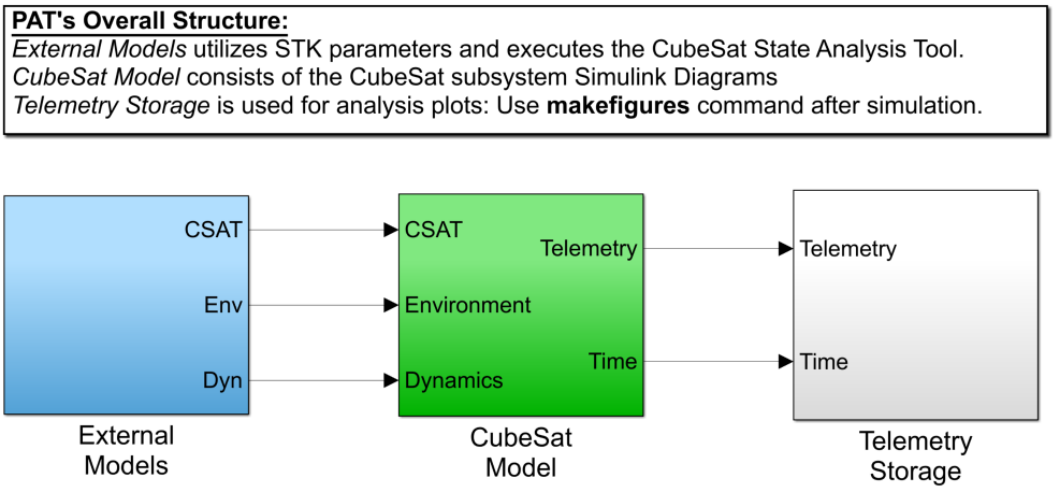

Figure 10 - PAT Simulink Structure [16]

CSAT is what powers the various Stateflow diagrams through the input and output variables. The inputs come from the STK reports that are then post-processed in MatLab. The variables present within the Integration Script allow for Simulink to execute and conduct the preliminary analysis of the EPS and ADCS subsystems. The EPS and ADCS subsystem analysis results are then input into the Stateflow model to determine if the fault modes are triggered,

necessary for the ultimate output back into Cameo. Figure 11 is a depiction of the CSAT in Simulink:

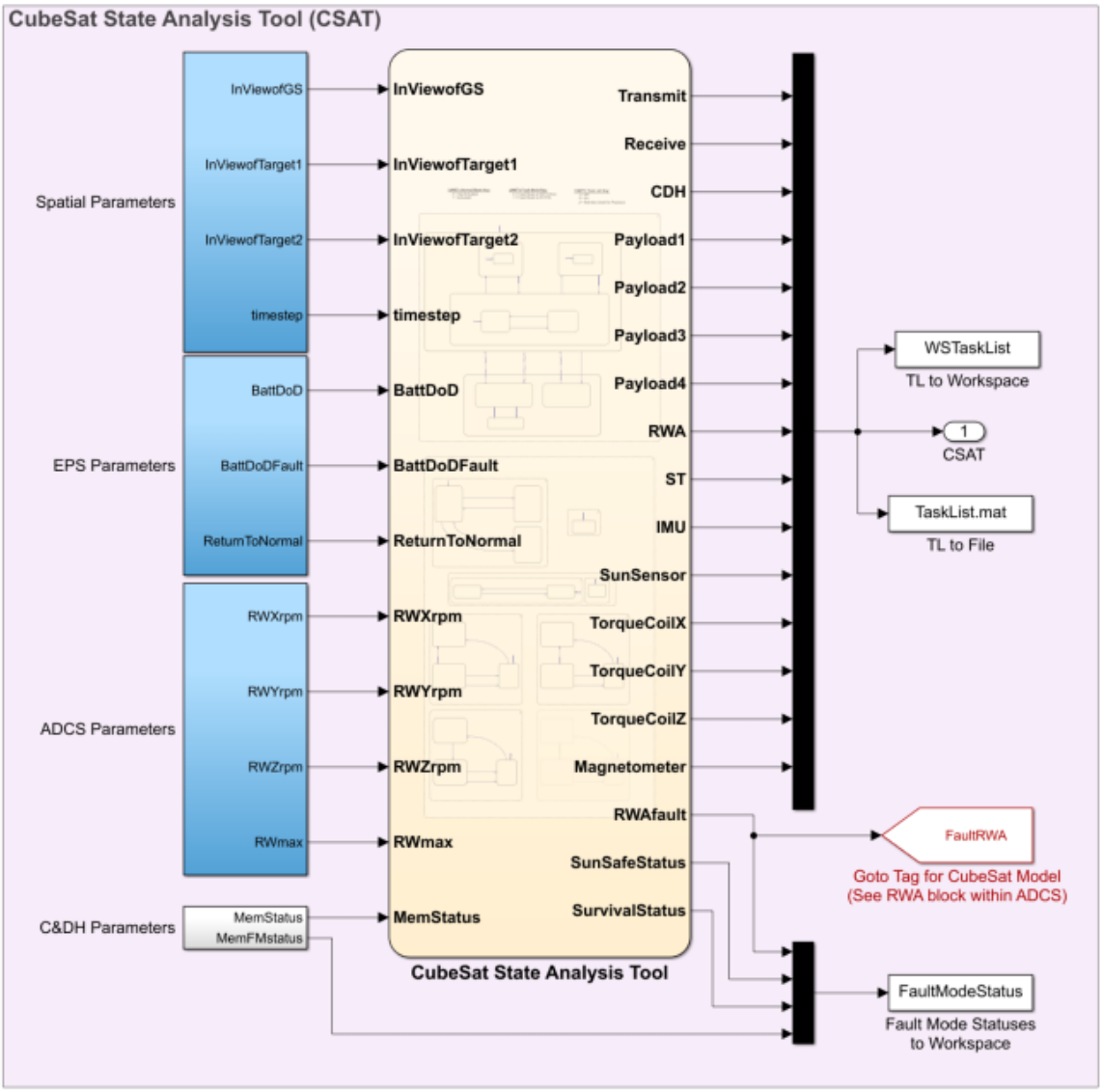

Figure 11 - CSAT Model in Simulink [16]

The Spatial Parameters block takes information from the STK Ground Station Access Report to tell the model if the satellite is in view of a ground station or target. The result acts as a Boolean, which is then used in the CSAT Stateflow model to determine which mode the satellite is in: Optical, Detection, BusOperations, Sun Soaking, Contacting GS, etc.

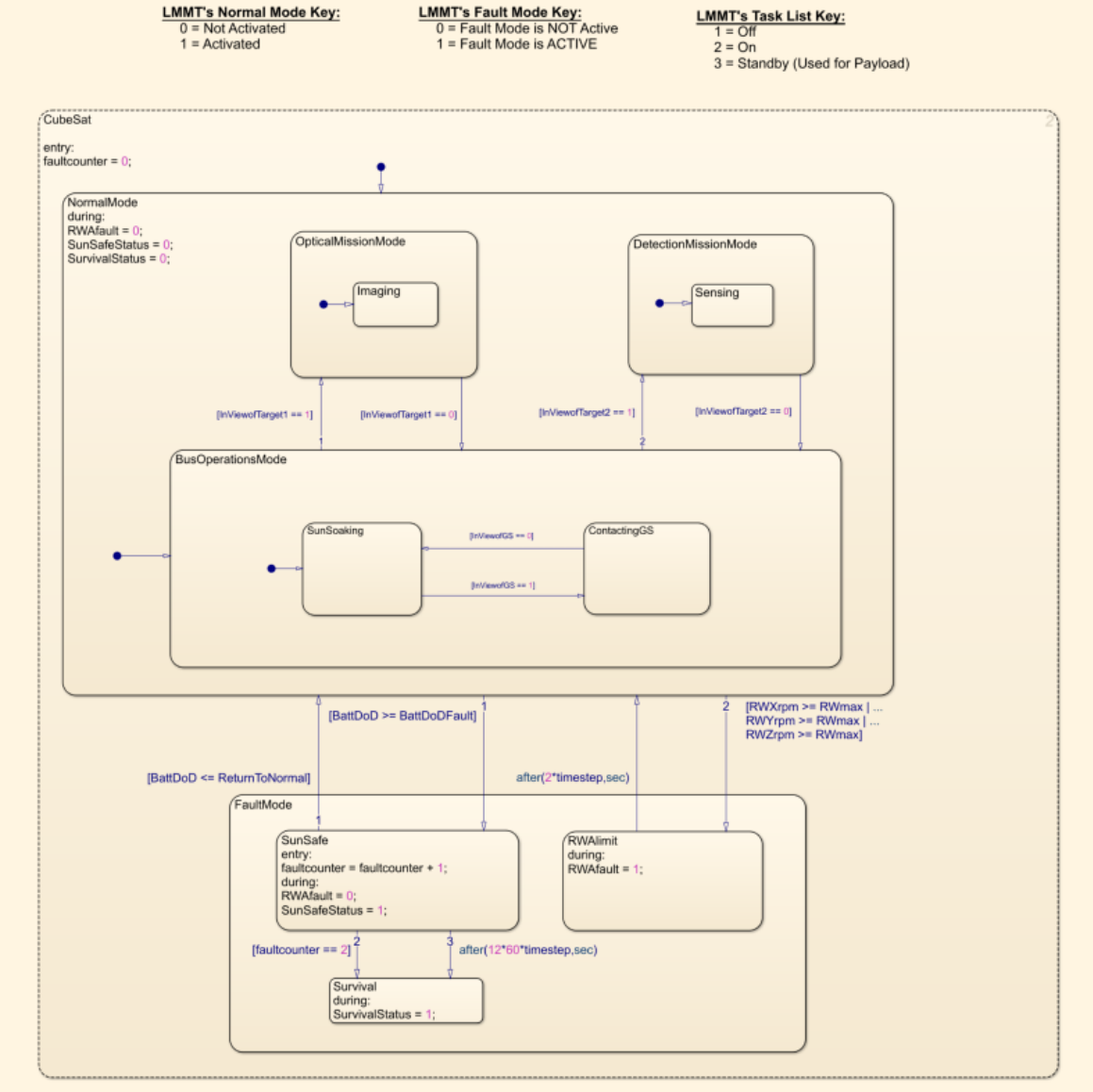

Figure 12 - CubeSat Stateflow Model [16]

The second part of the CSAT Stateflow model (Figure 12) is the Subsystem

Configuration State as seen below split into two figures for ease of viewing in Figures 13 and 14.

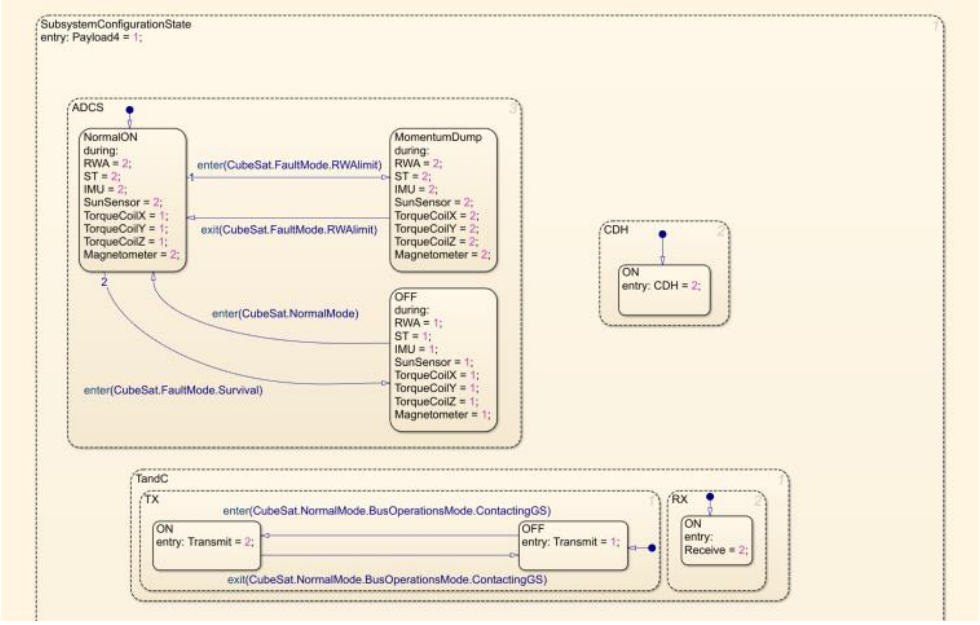

Figure 13 - Subsystem Stateflow Model, Part 1 [16]

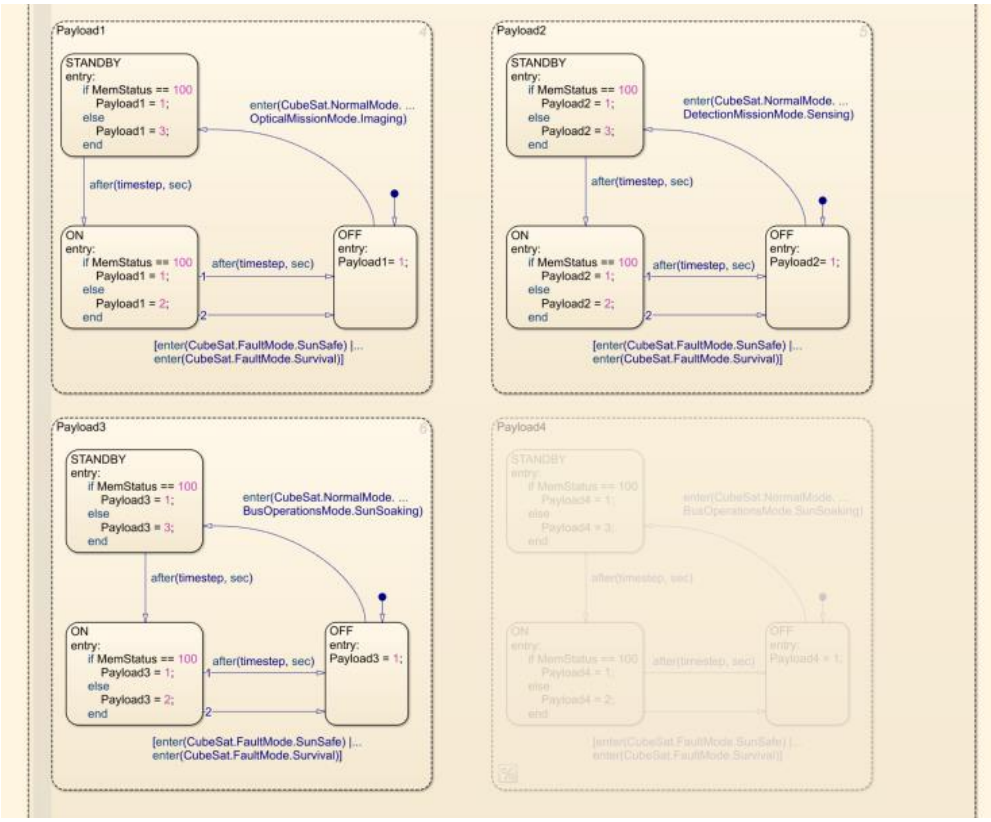

Figure 14 - Subsystem Stateflow Model, Part 2 [16]

External Models also includes the STK input parameters as seen below from the Environmental Model and the Dynamics Model. These are represented as ‘Env’ and ‘Dyn’ input parameters in the Simulink model.

The Environmental Model takes STK’s provided Moon and Sun Position, as well as the time to provide the spatial parameters of the CubeSat [35]. The Dynamics Model takes STK’s provided ephemeris data to calculate the position and velocity vectors. It also takes STK’s satellite attitude data to calculate the quaternion. This information is then used by the CubeSat Model to output subsystem information regarding the command, data, and power as well as ultimately calculating telemetry and time data. The telemetry and time data output from the CubeSat Model is then used as input for the Telemetry Storage Simulink model. The Telemetry Storage data is then used for the output of analysis plots.

The CubeSat Model (Figure 15) includes each CubeSat subsystem which can be expanded to show multiple Simulink models depicting the inter-workings of the subsystem to output the subsystem information regarding command, data, or power. Below is the top-level view of the CubeSat Model. By opening each subsystem, the various Simulink models will be displayed.

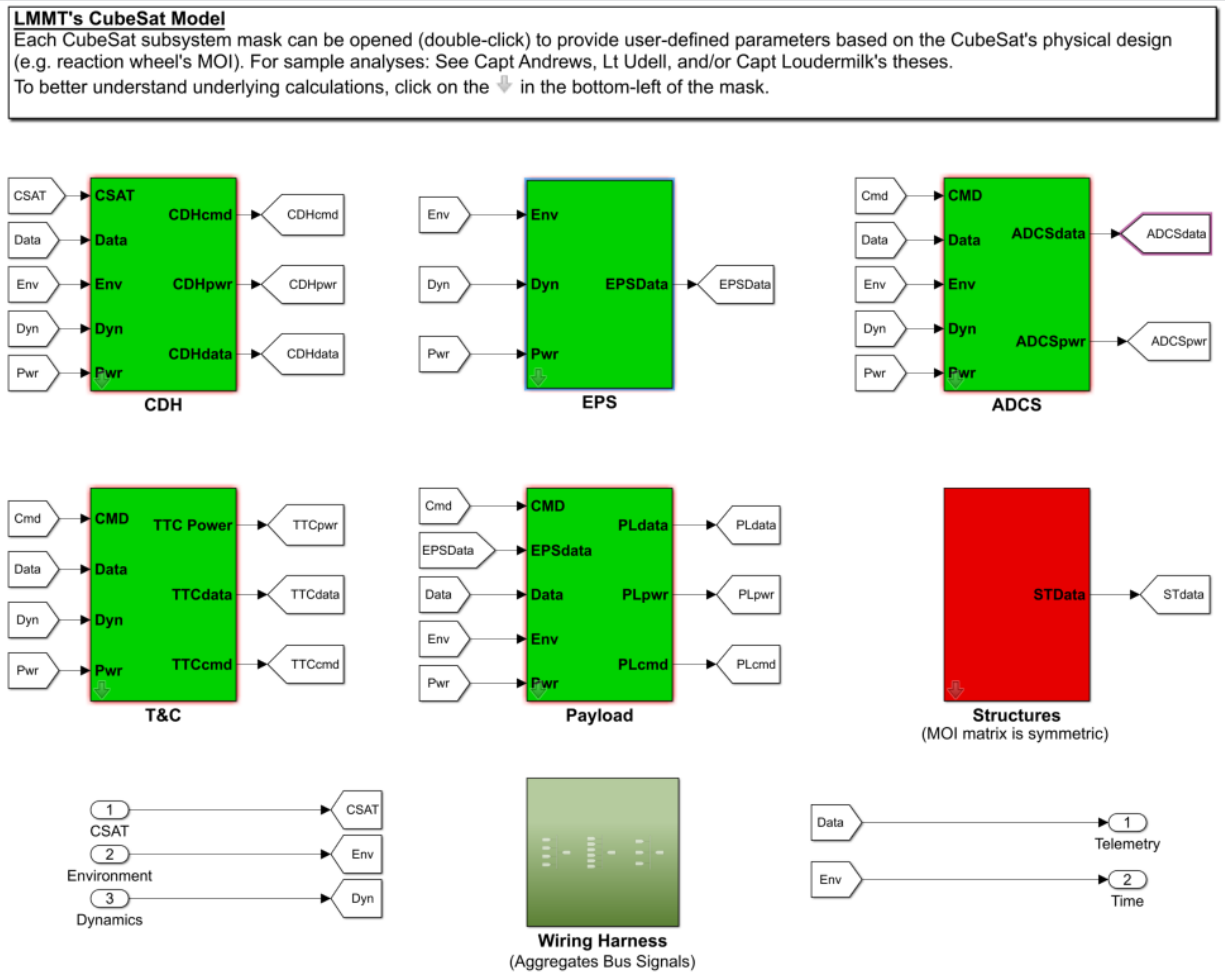

Figure 15 - LMMT CubeSat Simulink Model [16]

To execute the PAT, three MatLab files must be housed in the same directory named: *IntegrationScript.m*, *makefigures.m*, and PAT which is the Simulink/Stateflow model. The entire scenario is executed through Cameo by calling the *IntegrationScript* in the Payload Analysis Parametric Diagram. The PAT Users Guide has very specific details on how to run the simulation. The PAT was a vast improvement from the LMMT due to the user operating the entire simulation through one platform, using one file.

Upon execution, data is collected for the payload power and payload data rate. The data rate results are displayed as the bytes/sample time over the lifetime of the scenario (EpSec). The

data rate is shown for each payload individually and as a combined payload data rate. The payload power results are displayed in watts over the lifetime of the scenario (EpSec). The payload power is shown for each payload individually and as a combined payload power.

Concurrently with Brown, Capt Sean Kelly worked on additional facets of the CubeSat Reference Architecture (CRA) and MMT [37]. The CRA was used within Brown's Cameo model as a baseline generic CubeSat architecture [16]. The CRA holds all diagrams necessary to house the requirements, structure, and behaviors to conduct analysis on a generic CubeSat [37]. A user is able to take the CRA with its best practices, component library and example builds, and easily tailor it for their specific mission and design to enhance rapid development of their CubeSats [37]. Building out a useable MBSE CubeSat architecture provides a platform to accomplish digital engineering for a variety of CubeSat missions at AFIT, though AFIT is not alone in its mission to create a high-fidelity CubeSat model.

2.6 Related and Concurrent Work

Students at Cal Poly have been working on integrating MBSE, specifically SysML, into their existing Horizon Simulation Framework (HSF). They created two models, the Aeolus which was designed to "validate the concept that a SysML model could be translated into an HSF system script for use in HSF," and the ExoCube model which was created to apply the research to a real-world design scenario [38]. The models are built within Cameo, showcasing many of the mission operations and subsystem relationships of the CubeSat through bdd, ibd, state machine diagrams, and allocation matrices to name a few. The model also has a requirements satisfaction table, explaining which component satisfies the specific CubeSat requirement [38]. The SysML model was able to be linked to HSF, producing target data, power data, data rate information, and allowing for the simulation process to become streamlined using

MBSE [38]. Johnson continued Luther's work at Cal Poly, looking into more specific mission modeling for the CubeSat Astronomy Network [39]. Results were acquired for a single satellite and a network of satellites, describing observation capabilities throughout the mission, tracking multiple subsystem capabilities and tracing it back to mission and system requirements [39].

Analytical Graphics Incorporated (AGI) has created a digital engineering software tool called Moxie, used to facilitate SysML behavior modeling to validate the digital prototype [40]. Moxie takes the states defined in a state machine, and then executes those states within a physics-based simulation environment [40]. Using Moxie removes the step of scripting through something like MatLab, and completely runs the simulation through the state machine to a system like STK. After the simulation, Moxie provides a number of reports that one can use to validate the state model and support debugging [40]. See Figure 16 for a visual representation of Moxie:

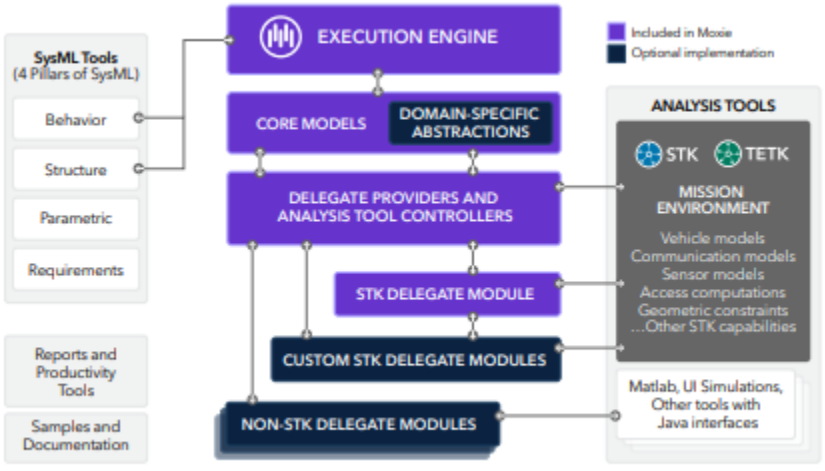

Figure 16 - Moxie Visual Overview [40]

2.7 Summary

In summary, Chapter I laid out the research objectives to integrate a propulsion system into the current CubeSat design, in order to validate payload requirements and determine the suitability between differing propulsion systems. Chapter II further explained key concepts that are necessary to understand how the research objectives will be accomplished, as well as related and current work in the area of CubeSat mission modeling using MBSE. Chapter II discussed the foundational elements of MBSE and how it is implemented in this thesis through the use of various SysML diagrams. Specifically, CubeSat missions can be modeled through the use of MBSE enhancing verification and validation of requirements, in this case AFIT's Grissom-2 mission. Chapter III explains how to evolve the PAT to include a propulsion system, which enables various mission options to be verified based on the requirements set forth to further enhance payload analysis and now also include thruster analysis.

III. Methodology

Chapter III outlines the steps taken to add a propulsion subsystem to the PAT to allow for efficient and accurate modeling and simulation of the Grissom-2 mission when it includes a thruster. Significant updates were made to the *IntegrationScript* within MatLab to create the new STK scenario using Astrogator. Updates were also made to the Cameo model, as Cameo switched overto CATIA. Propulsion updates were made using the propulsion template created in the CRA. Within CATIA, many blocks were added to the diagrams to accurately model the propulsion subsystem, as well as value properties to run the scenario through the PAT parametric diagram. Finally, updates were made to the Simulink and Stateflow models to include a propulsion subsystem and show the state changes the CubeSat experiences as it thrusts throughout its mission. Contrary to Brown's methodology, updates were made to move the PAT from a static model to a quasi-static model through the use of the Simulink and Stateflow models, which define when a repropagation of the scenario is necessary to better define the complex logic of a propulsion scenario. The scenario was then used to conduct a propulsion trade study, which is described in Chapter III and the results are found in Chapter IV.

3.1 Propulsion Scenario

The entire STK scenario is scripted within MatLab and required significant updates from the original satellite scenario used for the PAT, as described in Chapter II. The main additions came from adding a thruster and simulating an orbit raising scenario. To simulate satellite maneuvering, STK's Astrogator was used. Astrogator allows for the satellite to perform specified maneuvers throughout the scenario, utilizing STK's calculations to provide realistic maneuvering without predefining all the maneuver events a priori. For example, Astrogator allows for the user to specify in STK where the satellite needs to maneuver to and STK determines what the bes

thrust direction and duration should be. The user does not need to perform any calculations or physics-based diagraming to achieve a realistic satellite maneuver simulation. Below is a portion of the script used to set Astrogator as the satellite propagator.

Set-up Astrogator

```
%Change propagator to Astrogator
    satellite.SetPropagatorType('ePropagatorAstrogator')
        % Note that Astrogator satellites by default start with one Initial State
        % and one Propagate segment

        % Create a handle to the Astrogator portion of the satellites object model
        % for convenience
    ASTG = satellite.Propagator;
        % Create a handle to the MCS and remove all existing segments
    MCS = ASTG.MainSequence;
    MCS.RemoveAll;
```

Figure 17 – MatLab script for setting up Astrogator

After Astrogator was set as the propagator, the Astrogator maneuvers that were added are listed in Table 5 as follows:

Table 5 - STK Orbital Phases

Maneuver	Description
Inner Orbit	Initializes state data. Initial values are based on Keplerian elements and have the following parameters: - Semi-major Axis: 6828.1 km - Argument of Perigee: 0 - Eccentricity: 0 - Inclination: 60 degrees - RAAN: 0 - True Anomaly: 0
Propagate	Propagates the satellite until the stopping conditions are met. Propagates using the Earth HPOP Default v10 propagator. Stops propagating when the satellite altitude reaches 425 km.
Start Transfer	Sequence that runs the targeting profile. Includes the first burn which maneuvers the satellite with an impulsive burn, using the Earth HPOP Default v10 propagator.
Transfer Ellipse	Propagates the satellite until stopping conditions are met. Propagates using the Earth Point Mass propagator. Stops propagating when the satellite reaches apoapsis.
Finish Transfer	Sequence that runs the second targeting profile. Includes the second burn which maneuvers the satellite with an impulsive burn, using the Earth HPOP Default v10 propagator.
Final Orbit	Propagates the satellite until stopping conditions are met. Propagates using the Earth HPOP Default v10 propagator. Stops propagating when the satellite altitude reaches 425 km.

To increase fidelity of the model, modifications to the actual satellite were also made. These modifications can be split into modifying the actual satellite and modifying the thruster model. When a generic satellite is first created in STK, the system automatically chooses specific parameters, simulating a much larger satellite and thruster than the desired 6U CubeSat. The following parameters were modified within the MatLab script to create a satellite that more closely resembles a CubeSat:

Table 6 - STK Spacecraft Parameters

Spacecraft Parameter	**Default Satellite**	**Grissom-2 CubeSat** [11]
Dry Mass	500 kg	9.1490 kg
Drag Coefficient	2.2	2.2
Drag Area	20 m^2	0.2 m^2
Solar Radiation Pressure Coefficient	1	1
Solar Radiation Pressure Area	20 m^2	0.2 m^2
Radiation Pressure Coefficient	1	1
Radiation Pressure Area	20 m^2	0.2 m^2

These changes are necessary to gather accurate information regarding orbital perturbations as the scenario runs. The solar radiation pressure affects the acceleration of the satellite, which impacts how quickly the satellite experience orbital degradation. The radiation pressure influences the radiation pressure models, simulating the radiation reflectivity of the satellite based on the area of the satellite exposed to radiation pressure. Using Astrogator, the Earth HPOP Default v10 propagator was used, except for during the transfer ellipse where the Earth Point Mass propagator was used. The propagation of a satellite is largely affected by its mass and drag area; therefore, to accurately depict the lifetime of the satellite, these properties must be realistically included in the model. The scenario did alter the starting semi-major axis slightly. STK's radius of Earth is 6378.1 km, meaning that for an altitude of 450 km, the semi-major axis would need to be 6828.1 km (under the spherical Earth assumption).

To account for the added thruster, specific to the 6U, a cold gas thruster was added to the model using the custom engine feature in STK. The custom engine was scripted using the following parameters in Table 7:

Table 7 - STK Engine Parameters

Fuel Tank Parameters	**Default Satellite**	**Grissom-2 CubeSat**
Fuel Density	1000 kg/m^3	1.23 $x10^{-9}$ kg/m^3
Fuel Mass	500 kg	0.851 kg
Isp	300 s	40 s
Thrust	500 N	25 mN

Other cold gas thrusters could be modeled using this same custom engine by adding modifications to the Isp, thrust, number of thrusters, fuel mass, dry mass, and power levels of the different thrusting states. While using a cold gas thruster, it was assumed that the maneuvers would be impulsive instead of continuous. To further increase the fidelity of the model, other engine parameters such as tank pressure, tank volume, tank temperature, and maximum fuel mass may be added to the engine model.

All STK reports generated from Brown's scenario are generated in the new scenario, to include the addition of the LLA position report, and a maneuver report that includes each maneuver's start time, stop time, delta-V and fuel used. The LLA position report was used in determining the altitude of the satellite to know when a burn is necessary. The maneuver report was used to determine the thrust durations to be input into the CSAT. To simulate a propulsion subsystem, a generic heating duration of 1800 seconds was assumed as mentioned in Section 1.4. The heating duration can be changed within the CATIA model depending on the parameters of

the thruster being simulated. The duration of the burn was taken from the STK maneuver report which produces an estimated burn duration based on the specified thruster parameters.

Once the scenario was performing as described below, the script was then split into three separate scenarios to create the desired quasi-static simulation. The following parameters were considered when creating the scenario: time, battery depth of discharge (DoD), altitude, contact with a ground station, and contact with either of the targets. The first scenario consisted of the satellite beginning at its desired orbit, as described above, and propagating until it reached an altitude of 425 km. At that time, the STK simulation would end, and the results would be pushed through the PAT in Simulink to further determine what occurred during the mission. The data from the PAT was then parsed into a table of parameters and filtered to find a scenario time that had a battery DoD of less than 20%, an altitude below 430 km, and was not in contact with a ground station or target. The first time where all of these parameters were true was then used as the time that the first burn should take place to begin orbital maintenance.

The second part of the scenario consisted of all parts of the first scenario and continued to the first burn and transfer ellipse. The scenario time determined from the first simulation was input into the script to give the ideal time to burn, and the scenario was run through the first burn, and propagating for 20,000 seconds following the first burn. The STK simulation then ended, and the results were pushed through the PAT in Simulink. The data from the PAT was then parsed into a table of parameters, similar to the first scenario. The data was filtered to determine the optimal time to complete the transfer ellipse after the first burn and begin the second burn. The second burn would ideally occur at apoapsis in order to save the most fuel. Other than altitude at apoapsis, the DoD and access to ground stations and targets were also taken into account. The DoD had to be less than 20% in order to have adequate power to

complete the maneuver and not transition into a fault mode. Also, the satellite could not be communicating with a ground station or collecting data from a target, because the mission objectives of the payload data take precedent over when the burn should occur. The time of the second burn also had to be after the time determined for the first burn, which seems obvious, but for the logic to work on the table, it had to be specified. The scenario time for the transfer ellipse was then used in the third scenario to complete the execution of the second burn.

In the third scenario, all of the components of the first two scenarios were included. Additionally, a script to enact a second burn to return the satellite to nearly zero eccentricity after the first burn was included. After the second burn, the satellite then began propagating until it reached 425 km. The solution produces a time at which a second burn sequence would likely occur.

The combination of the three scenarios (Integration Scripts 1-3) enables a quasi-static scenario to be created. Multiple iterations of these three scenarios would need to be repeated in order to create a full mission scenario. A visual representation of the three scenarios that make up the propulsion maneuver can be seen in Figure 18.

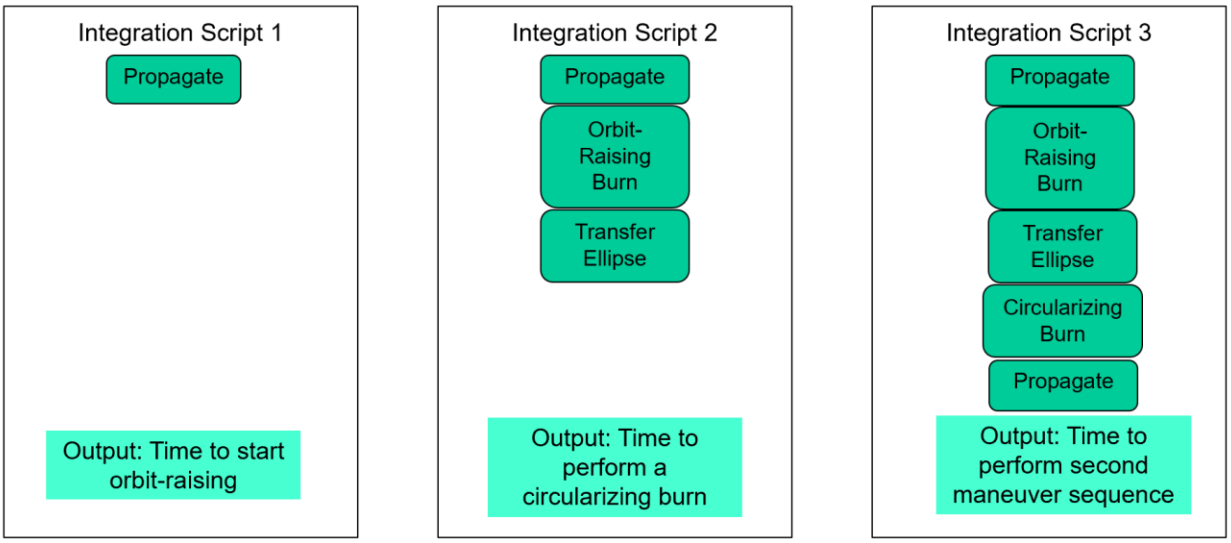

Figure 18 - Propulsion Scenario

3.2 Integrating Propulsion into CATIA

To run the STK scenario from CATIA, all of the necessary variables for the propulsion scenario need to be input into the CATIA model. In Brown's model, there was no place-saver for a propulsion subsystem; therefore, everything involving a propulsion subsystem needed to be added from the CubeSat Reference Architecture The containment tree in Figure 19 shows where the propulsion blocks were found and replicated for use in the Grissom-2 model.

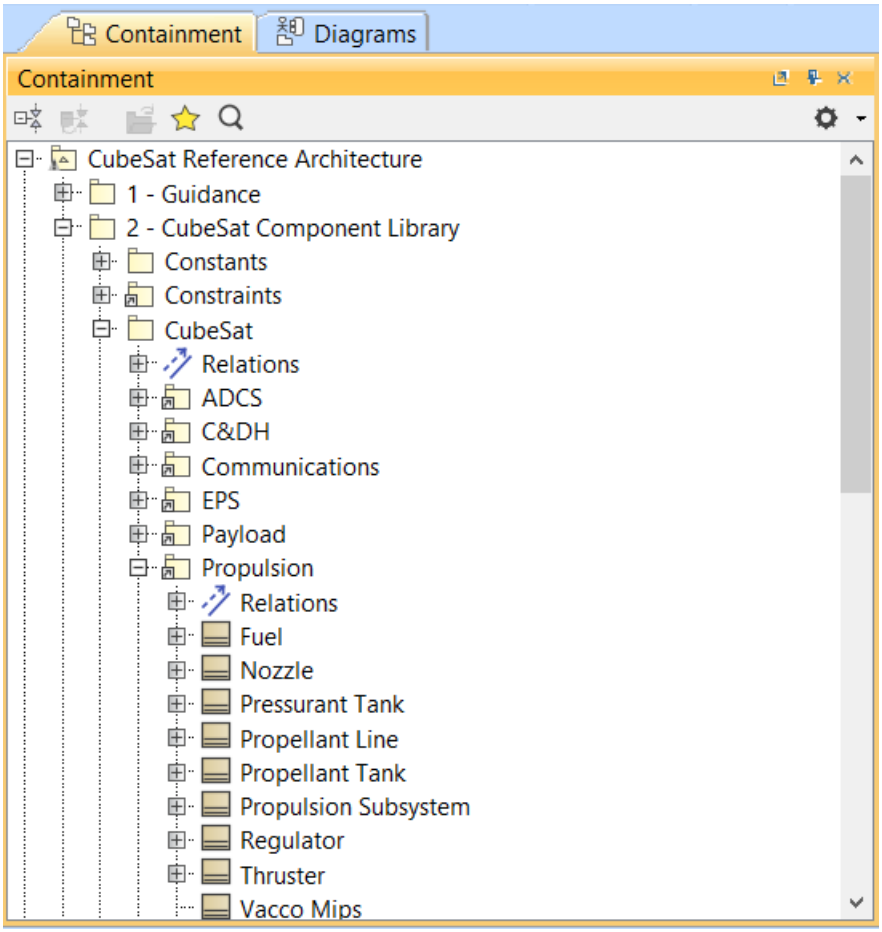

Figure 19 - CubeSat Reference Architecture Propulsion

The blocks from the CRA are then used to add a propulsion component to the 2-Structure package within the CubeSat package and within the CubeSat bdd in Figure 20.

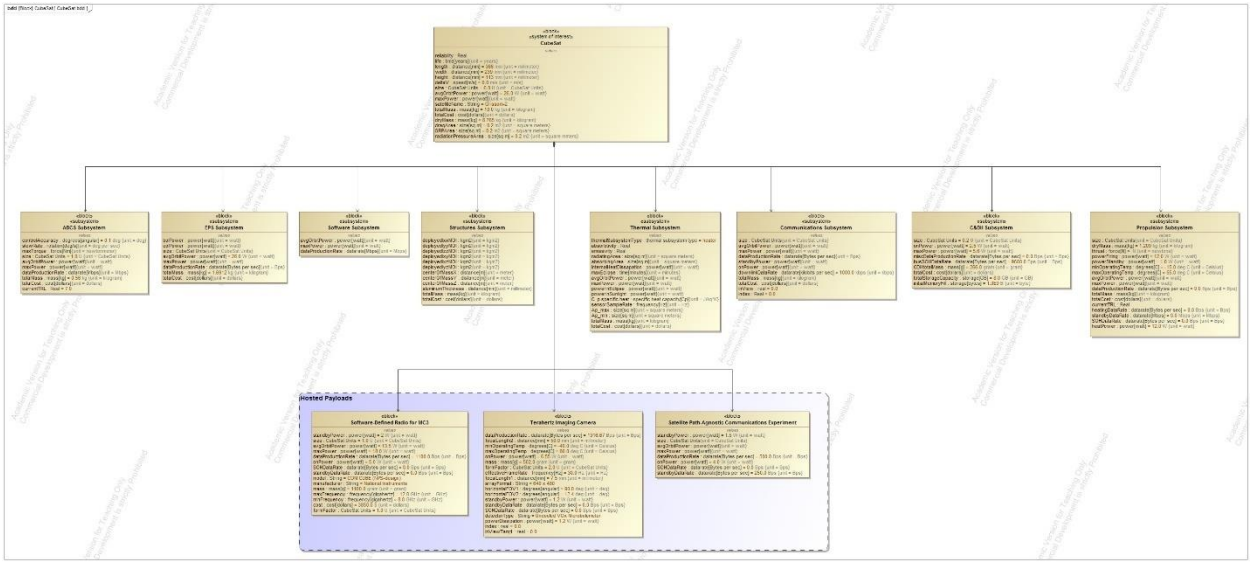

Figure 20 - CubeSat bdd

The various value properties of the propulsion subsystem are also added within the propulsion subsystem block. The propulsion subsystem block also has its own bdd where different components of the propulsion subsystem are included as blocks with a directed composition of the propulsion subsystem as seen in Figure 21.

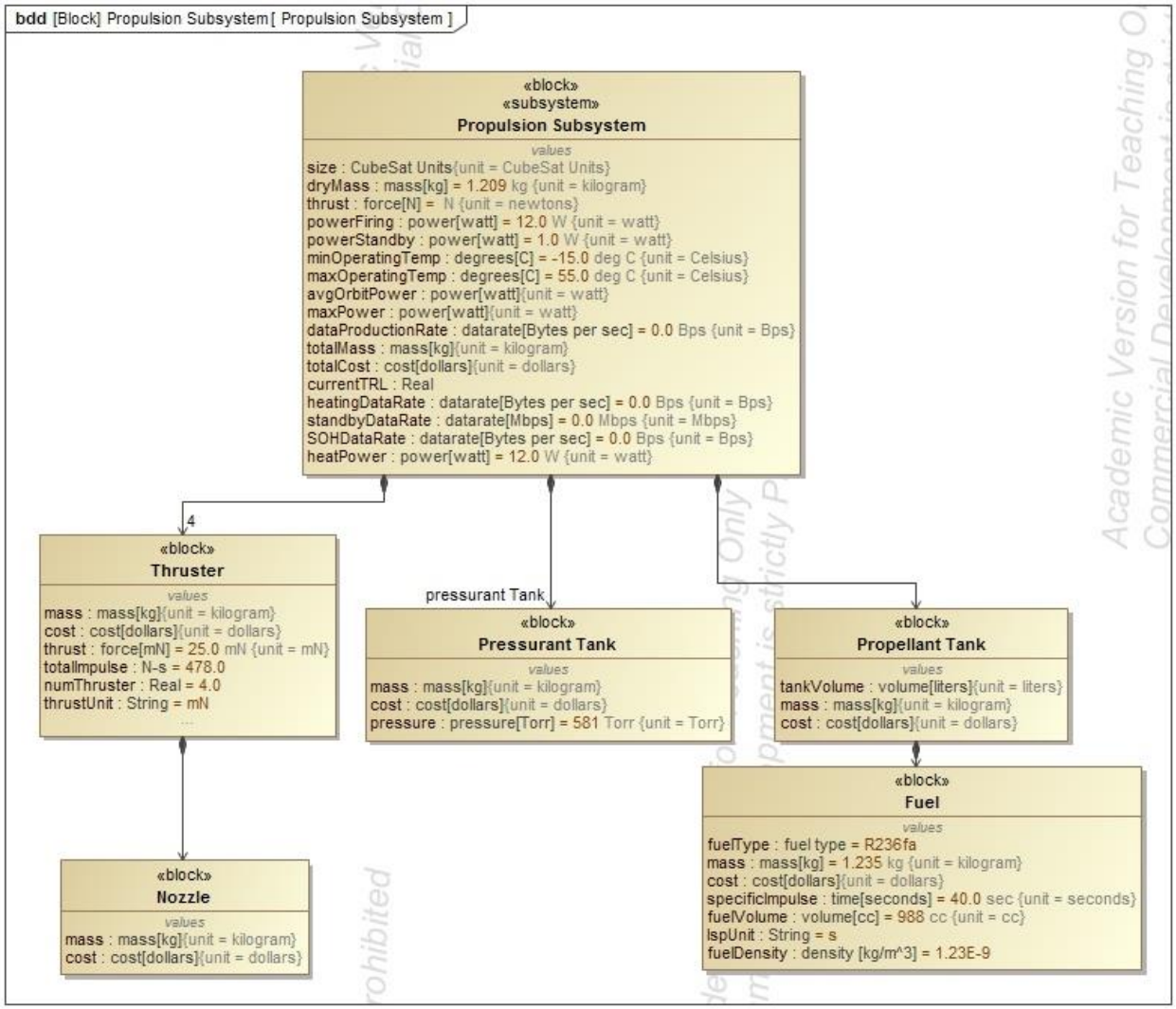

Figure 21 - Propulsion Subsystem bdd

The propulsion subsystem block is then reused throughout the model in the Mission Context bdd, Simplified Mission Context bdd (Figure 22), and the Mission Context ibd. Within the Mission Context ibd, additional operations are also added to the CubeSat part property to account for operations completed by the propulsion subsystem, to include thrust and heat fuel.

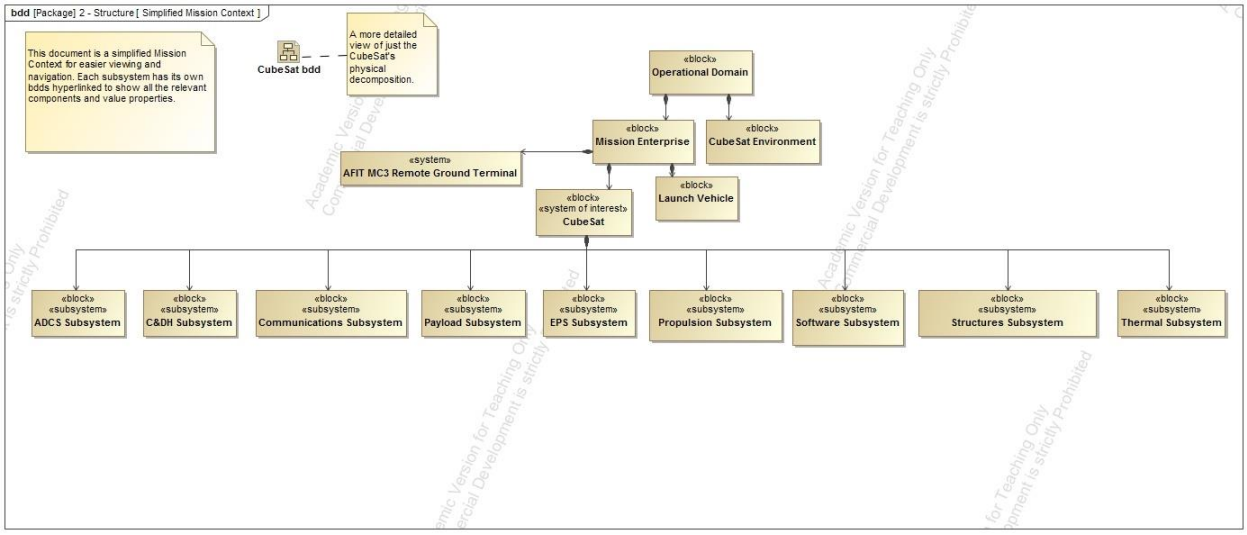

Figure 22 - Simplified Mission Context

Within the Behavior package, a propulsion activity was added to the CubeSat Mission Activity Decomposition bdd. Within the Subsystem Activities bdd, there is a Propulsion Subsystem that has the activity “Generate Thrust”, but other maneuvering activities are added in order to build out the propulsion subsystem activity diagram.

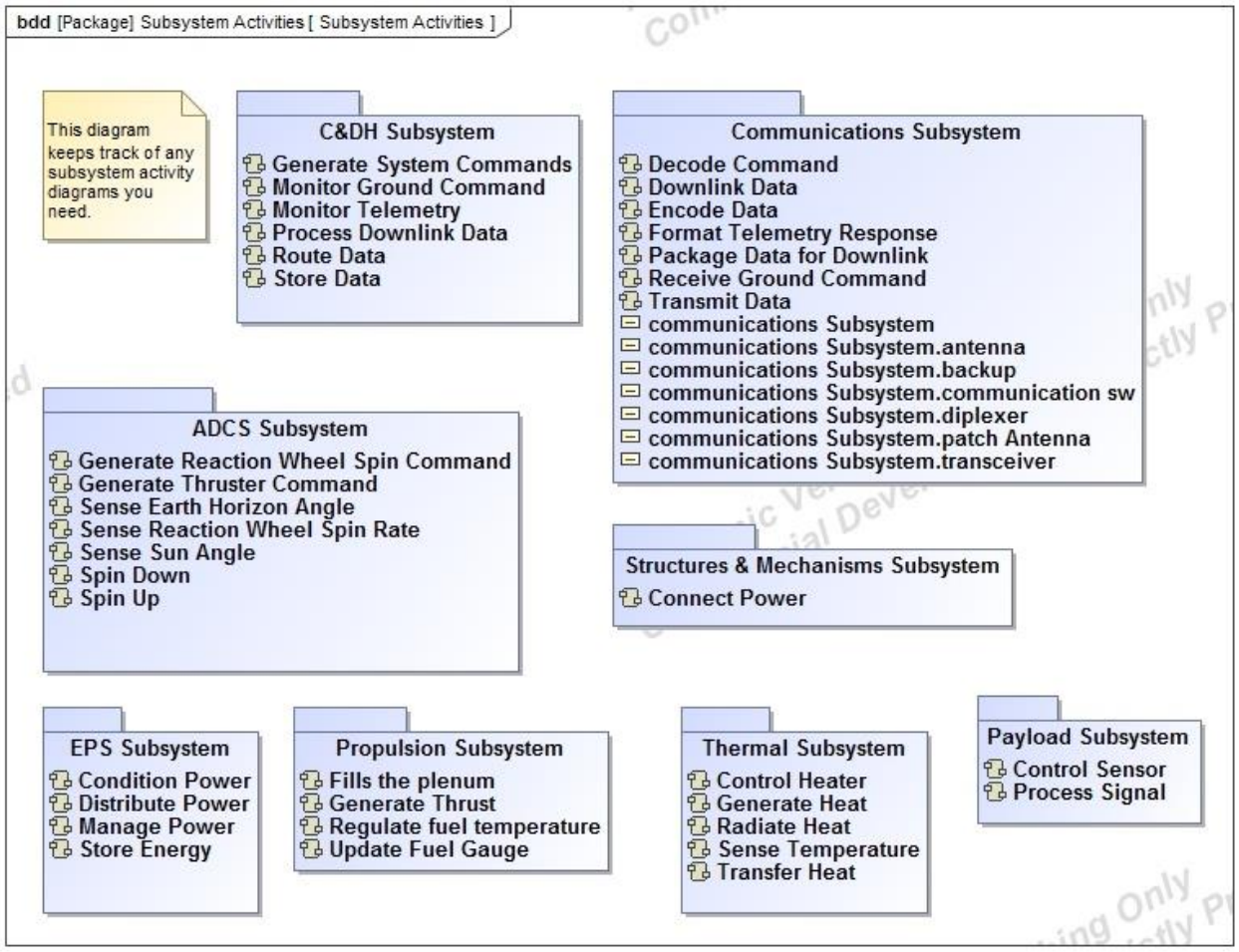

Figure 23 - Subsystem Activities bdd

For the Analysis package, all of the updates for a propulsion subsystem occur in the Verification Analysis package. For the Link Budget Analysis, because the altitude is changing as the satellite propagates, additional analysis will need to be conducted to see if there are specific altitudes where the Link Margin is unacceptable. A Link Margin Analysis will not be completed as part of this thesis. The same is true for the Image Quality Analysis within the Optical Analysis Package; with a changing altitude, there may need to be additional analysis to calculate the GSD throughout the mission life of the CubeSat, as well as the Orbit Analysis within the Orbit Analysis package.

The Payload Analysis Tool package is where a majority of the CATIA updates took place in order to integrate with the MatLab script to run the PAT. The STK Scenario Parameters part property changed the scenarioStopTime value property from a one-week mission to a mission that lasts the duration of the satellite maneuver, which is estimated to be two months based on the simulation run within STK. The Orbit part property determines the initial orbit of the satellite, and therefore all value properties remain the same.

For the CubeSat Bus portion of the parametric diagram, all subsystems remain the same; there are no component changes outside of the addition of a propulsion subsystem. With that said, the propulsion subsystem part property is added to the model, similar to how the C&DH subsystem part property is included. The value properties of the propulsion subsystem were added to the parametric diagram and connected to their specified constraint parameters of the Payload Analysis Tool constraint property as seen in Figure 24. The constraint parameters are then labeled with the same variable names needed for the MatLab script of the STK scenario. The specified payloads do not change from Brown's previous work.

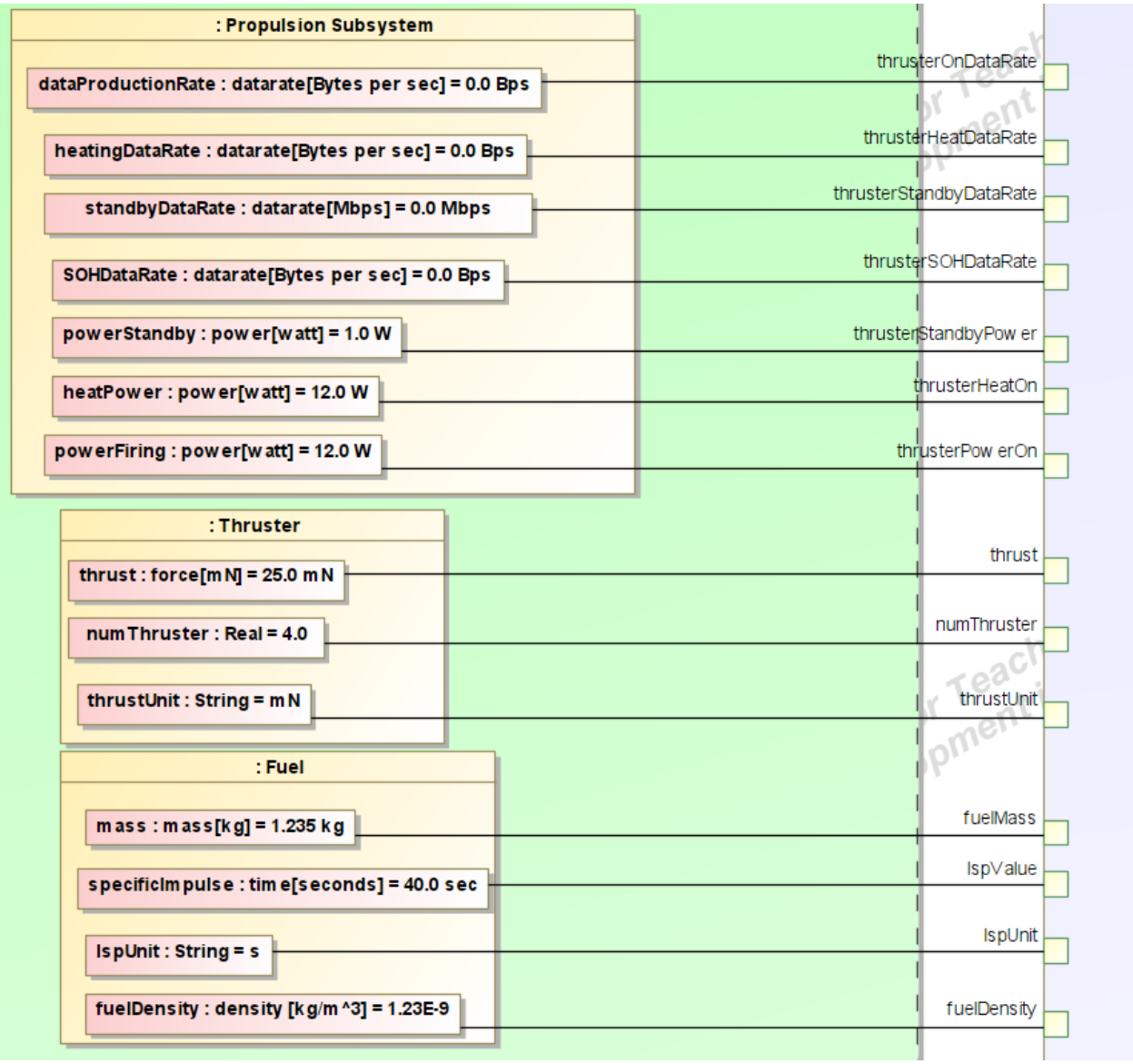

Figure 24- Propulsion Value Properties

Looking into the right side of the Figure 8 Payload Analysis Tool parametric diagram, the Analysis Context part property now includes value properties associated with the mission implications of having a propulsion subsystem such as mission life, delta V, and fuel used. These value properties will also be shown in the Instance Table.

3.3 Integrating Propulsion with Simulink

The majority of the changes made to the Simulink model were due to the addition of a propulsion subsystem. Additional changes were made to the payload model based on input from Erbe and Lemmer [41], which is discussed below.

Starting with the CSAT, found within the External Models portion of the Simulink PAT, a "Thrusting" input was added to the model. The thrusting values were taken from MatLab, calculated based on when the thruster would be in standby, heating, and performing a thrusting maneuver. The below script in Figure 24 is from the *IntegrationScript_3.m* file. The script varies slightly for *IntegrationScript_2* because there is not yet a DynamicPropStop2 variable.

```
heatingDuration = 1800; %s
heatingStart = dynamicPropStop1 - heatingDuration;
heatingEnd = dynamicPropStop1 - 60;

heatingRow1 = dynamicPropStop1;

for k = 1:size(tv,1)
    heating1 = find(tv < dynamicPropStop1 & tv > dynamicPropStop1 - heatingDuration);
    heating2 = find(tv < dynamicPropStop2 & tv > dynamicPropStop2 - heatingDuration);
    heating = [heating1 ; heating2];
    thrustTime(heating) = 2;

    thrustTime1 = maneuver(5,1);
    thrustTime2 = maneuver(5,2);
    thrusting = find((tv >= dynamicPropStop1 & tv < dynamicPropStop1 + thrustTime1) | ...
        (tv >= dynamicPropStop2 & tv < dynamicPropStop2 + thrustTime2) );
    thrustTime(thrusting) = 1;
end

% Finally, combine time and thrusting vectors to be used by CSAT
Thrusting = transpose([tv; thrustTime]);
```

Figure 25 - MatLab Script for Thrusting Data

The basis of the parsing of the thrusting data was taken from the code necessary to parse the access data for the ground station and targets. One assumption associated with the thruster was that it performs impulsive rather than continuous burns. An impulsive burn is theoretical because in reality the burn is not instantaneous (due to max thrust possible), instead it happens over time.

This proved difficult to calculate the time associated with the thrusting state, as the simulation assumes the burn duration is instantaneous. To account for how long the burn was occurring, the finite burn time calculated by the STK maneuver report was used for the time the satellite was spent performing the thrusting maneuver (thrustTime1 and thrustTime2). The analysis of burn times is present in section 4.4. Another assumption made was that the heating time for the thruster was set to occur for the 1800 seconds before each burn (heatingDuration). In reality, the heating time for each thruster would be different depending primarily on size and type of fuel and ambient temperature. Calculating the heating time unique to each thruster is an area of interest for continuing research which would increase the fidelity of this model. The thrusting data was then used within the "Thrust" state, that occurs when the satellite is in Bus Operations Mode. As seen below, the Thrust state is independent from the SunSoaking and ContactingGS state. When looking at the model, Figure 26, the satellite cannot perform a thrusting maneuver if it is contacting a ground station. When Thrusting equals one, the propulsion system is thrusting, when thrusting equals two the propulsion system is heating.

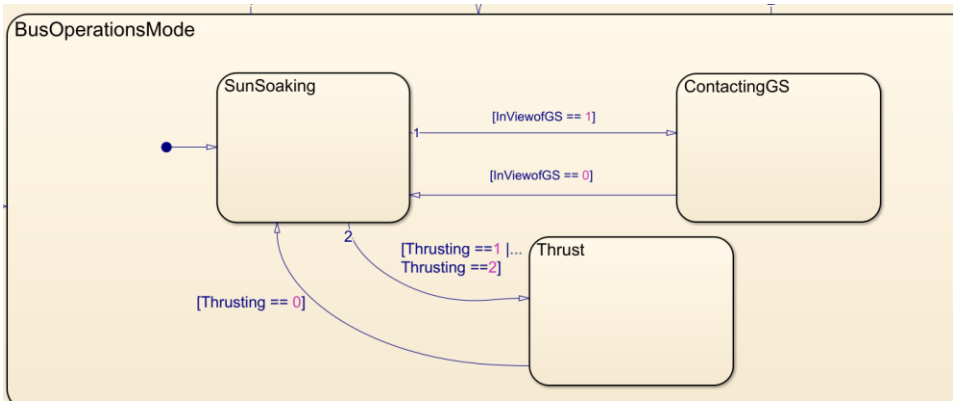

Figure 26 - Bus Operations Mode with Thrust State

When the satellite enters the thrust state, the specifics of the propulsion subsystem are processed using the below subsystem model in Figure 27.

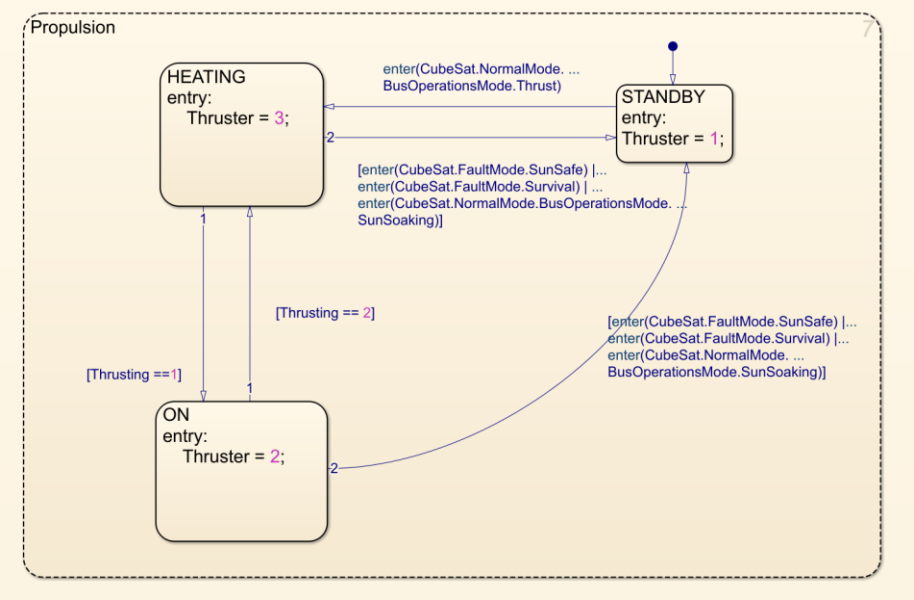

Figure 27 - Propulsion Subsystem Stateflow

There is no "off" state within the propulsion subsystem, rather a standby state takes its place. With the type of thruster being modeled, there is always a small power draw of at least one Watt; therefore, the thruster is never truly off and not drawing power. If a thruster is being modeled that has a true off state, this model can be altered to suit that reality. The propulsion subsystem will enter a standby state if the satellite is in a fault mode, or is no longer heating or thrusting, and therefore is in a sun soaking state. The data collected within the Stateflow CSAT will then be used in the CubeSat Model.

A separate propulsion subsystem was added to the CubeSat Model, along with a subsystem mask. The subsystem mask is shown in Figure 28 producing the thruster power and data rate as calculated by the CSAT. These values are aggregated with the other bus signals in the Wiring Harness as shown in Figure 29.

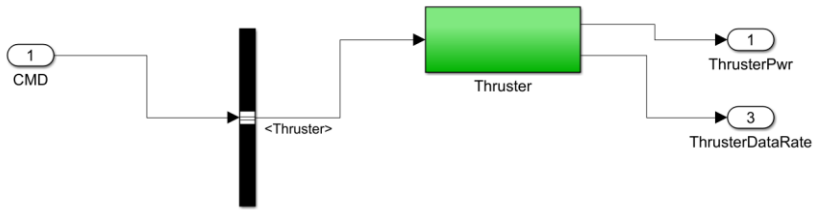

Figure 28 - Propulsion Subsystem in Simulink

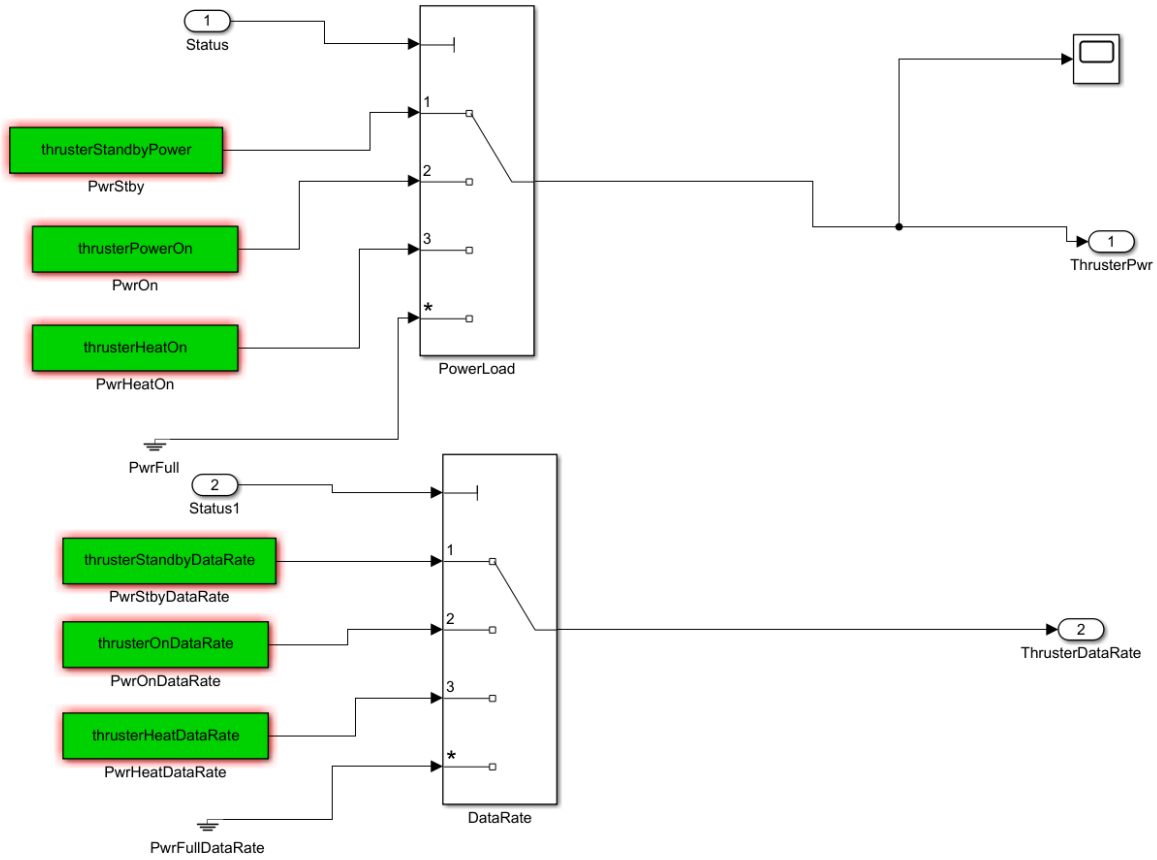

Figure 29 - Thruster Model for Power and Data Rate

The thruster power is then input as a power value in the EPS mask, adding as an input to the battery performance, which calculates the battery DoD and the component power total values.

Additional updates were also made to the Payload1-4 blocks of the Stateflow model within the CSAT. Brown's model of the Payload is shown in Figure 30:

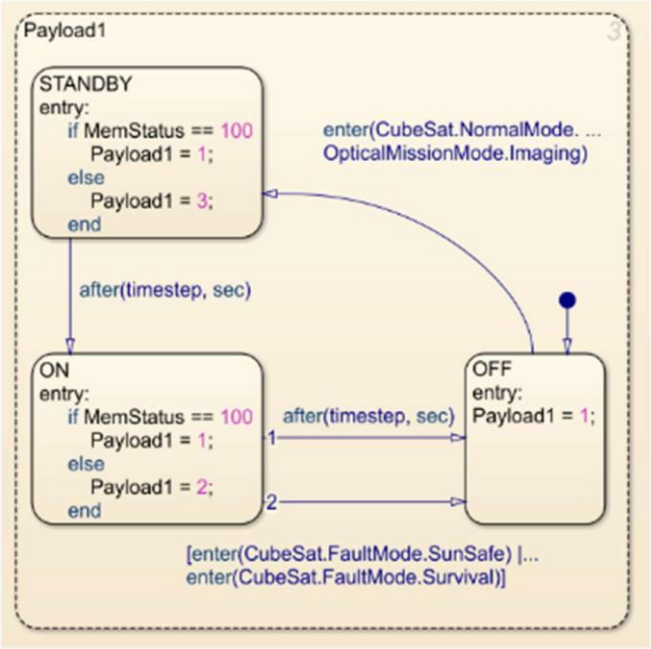

Figure 30 - Brown's Payload Stateflow Model [16]

This interpretation of the Payload demonstrates the payload cycling from off to standby to on, then back to off every time the target is out of view [41] . In reality the Grissom-2 payloads would not turn off in between target connections, the payload would merely turn into a standby mode until a target is acquired. The payload would only turn off when in a Fault Mode [41]. The following upgrades were made by Erbe and Lemmer, and were then updated for the PAT as seen below in Figure 31:

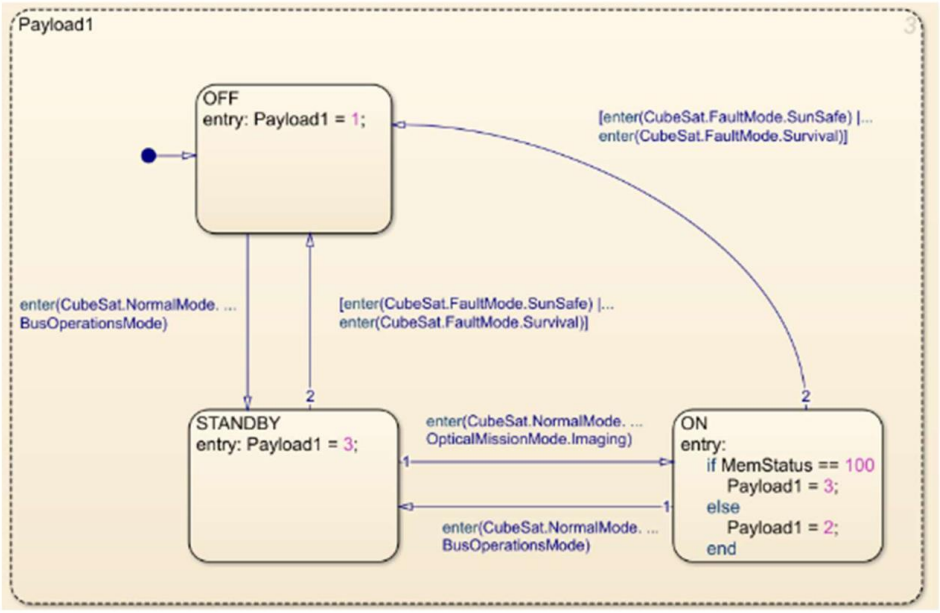

Figure 31 - Erbe and Lemmer's Payload Stateflow Model [41]

Using the updated model gives the model greater fidelity and provides more accurate information concerning the payload and its mission capabilities. To account for this change, Erbe and Lemmer also created a MatLab function to extract the In View State data from the STK data outputs. The function does not take into account fault modes and is therefore not used in the updated PAT. By updating each payload's Stateflow diagram, an accurate standby mode was achieved using the existing data provided by STK and parsed within Brown's MatLab code. Figures 32-34 are the updated Stateflow diagrams for payloads 1-3.

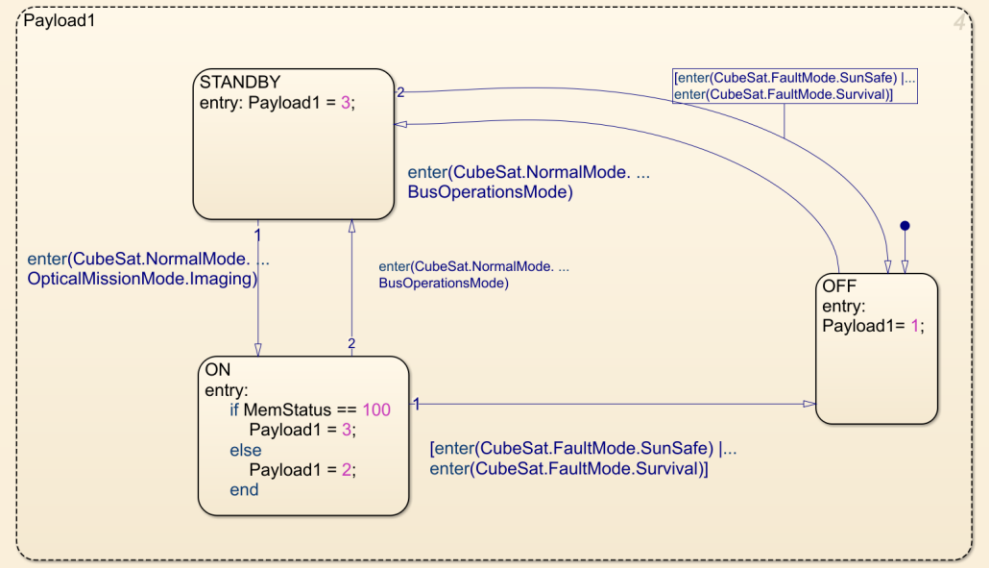

Figure 32 - Payload-1 Stateflow Diagram

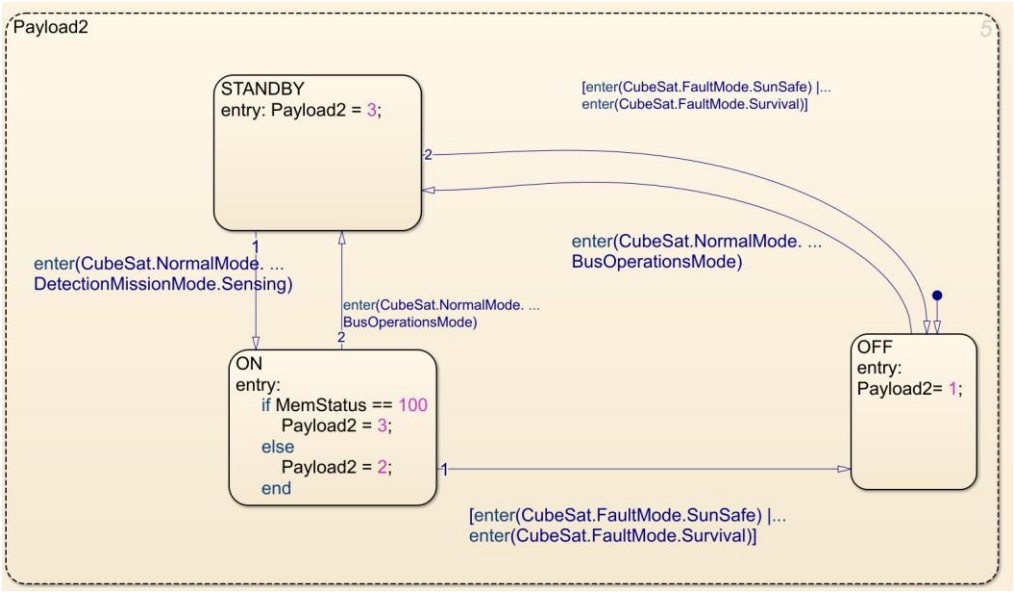

Figure 33 - Payload-2 Stateflow Diagram

Payload-1 and payload-2 are similar because each have a specific mission where the payload is pulled out of bus operations mode and into their specified mission modes.

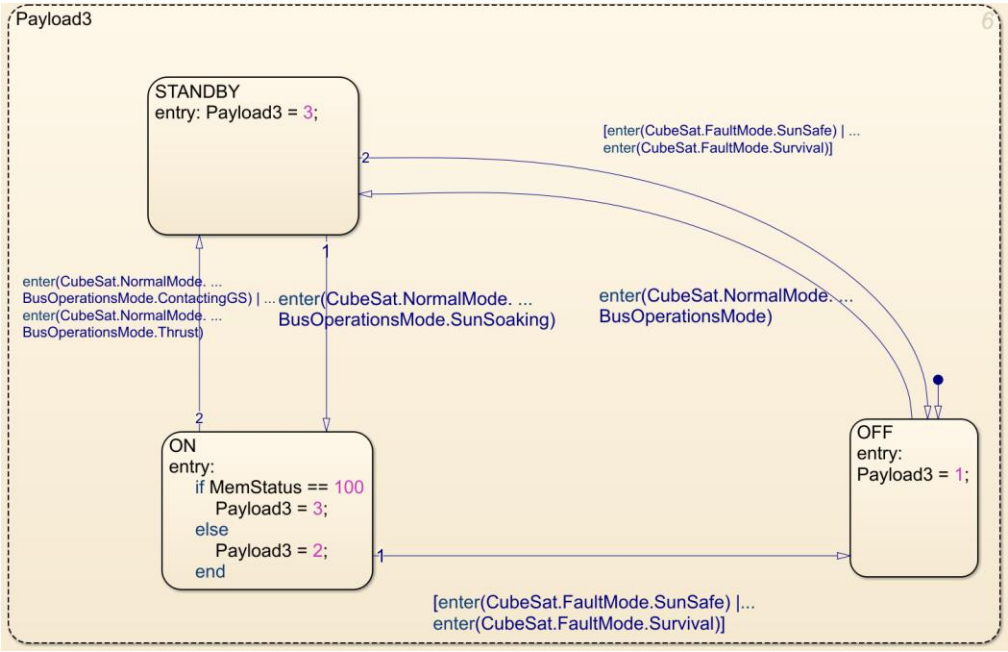

Figure 34 - Payload-3 Stateflow Diagram

Payload-3 is different from the other two payloads because it accomplishes its mission within the sun soaking state of the bus operations mode. When the satellite is contacting a ground station or thrusting, payload-3 moves into standby mode. As will be discussed in Chapter IV, the new Stateflow model of the payloads drastically impacted the overall power output of the satellite, specifically with regards to payload-3.

3.4 PAT Testing

3.4.1 Payload State Change Testing

The first PAT testing occurred with Erbe and Lemmer's changes to the payload subsystem state diagram and In View State function. Testing was conducted using Brown's scenario, only adding Erbe and Lemmer's scripting to the MatLab code and changing the Stateflow diagram to include the new standby transitions. The In View State a1_StateVector was also added to the payload subsystem mask as specified above. When conducting the test, the scenario was run using the updated PAT and compared to the results that Brown recorded. The

power totals, data rates, and memory were of main concern, as those are primarily affected by the payload state changes. During this test, the power levels for the thruster were set to zero. Even though the thruster is not needed during Brown's scenario, the propulsion subsystem still draws power passively even when not heating or thrusting and therefore the power levels need to be turned off as to not interfere with the PAT payload state comparison.

3.4.2 Propulsion Subsystem Testing

The second set of testing involves the propulsion subsystem. The timing of the thrusting in the STK scenario must align with what is being tracked within the CSAT. The thrusting will primarily affect the ADCS and EPS subsystems with regards to the analysis this scenario is collecting. Both of these subsystems directly influence the satellite's fault modes.

The ADCS subsystem faults are driven by reaction wheel speed. If the reaction wheel reaches its limit, it will enter a fault mode. The reaction wheel speed is determined by the ephemeris and attitude reports produced by STK. The scripting for collecting and parsing the ephemeris and attitude files was updated, so that the type of scenario could be altered and the data would continue to be parsed correctly even if the data changed sizes. To test if the ephemeris and attitude data was collected correctly and transformed into the appropriate ADCS data, the angular rates of the ADCS subsystem were analyzed at the time of the thrusting maneuvers.

With the addition of the propulsion system, the EPS subsystem will also be affected. The power generated from the thruster was added to the overall power of the satellite, which affects the total component power and battery DoD. Further analysis of these changes is expanded upon in Chapter IV.

As stated above, Brown's PAT demonstrates a static model of the Grissom-2 mission scenario. The updated model demonstrates a quasi-static model, consisting of three different iterations that build on one another to create the full orbit-raising scenario. The first iteration of the quasi-static model looks very similar to Brown's scenario. The satellite begins at the original altitude and propagates until it reaches an altitude of 425 km. The data produced by STK from the scenario is then run through Simulink/Stateflow, and the state machine then determines when an appropriate time to burn would be to return to the original altitude. The appropriate time to burn depends on the power available, if a payload is operating, if the satellite is contacting the ground station, and the altitude. Once the burn time is determined, the second iteration is run. The second iteration includes the initial propagation, followed by the orbit-raising burn and the transfer ellipse. The data produced from the second iteration is then run through Simlink/Stateflow, and the state machine then determines the appropriate time to end the transfer ellipse and execute the circularizing burn. The third iteration includes the initial propagation, the orbit-raising burn, the transfer ellipse, the circularizing burn, and the final propagation which is executed until the altitude reaches 425 km. All of the different steps in the scenario are executed based on the burn times determined in the first two iterations.

3.5 Propulsion Trade Study

The data from the complete quasi-static scenario is then analyzed with varying CATIA parameters to affect the outcome of the mission and compare alternative satellite designs. Specifically, the propulsion parameters are manipulated to conduct a trade study of multiple cold gas propulsion systems. The trade study consisted of five notional cold gas propulsion systems with the following attributes seen in Table 8 below:

Table 8 - Propulsion Trade Study Parameters

Thruster	Isp (s)	Total Thrust (mN)	Fuel Mass (g)	Dry Mass (g)	Standby Power (W)	Heating Power (W)	Thrusting Power (W)
#1	40	100	1235	1144	1	12	12
#2	40	150	1280	1263	1.1	55	9
#3	40	100	177	513	1	12	11
#4	40	200	600	644	0.25	5	5
#5	60	100	635	1430	1	20	4.3

The propulsion subsystem parameters which are held as value properties within CATIA were altered for each propulsion system and a new simulation was run. After the simulation was completed, the Analysis Content results were saved within the Instance Table. This process was repeated for each propulsion system taking part in this trade study. The Instance Table was then examined in order to determine how the results compared to the original mission requirements. In the analysis, it was noted where the mission was propulsion limited, and how the mission was improved through the use of a particular propulsion system.

3.6 Summary

In summary, Chapter III discussed how a propulsion subsystem was integrated into the PAT, as well as changes made to the model to incorporate updates to the payload state model. Based on the changes made and value properties (see Appendix A) used throughout the model, a propulsion trade study was created. The results of the model changes as well as the propulsion trade study are discussed in Chapter IV. The three updated *IntegrationScripts* can be found in Appendices B-D.

IV. Analysis

Chapter IV contains the results and analysis from the mission scenario testing. Adding the propulsion system to the Grissom-2 model, as well as changing the payload state diagrams produced a number of clarifying results. Chapter IV begins with describing the propulsion scenario and how the specific parts of the scenario were chosen to be used in the simulation. The second part of Chapter IV portrays the results of the propulsion trade study, to include which thruster is best suited for the Grissom-2 mission based on the recommended figures of merit. The last part of Chapter IV analyzes the payload state changes and how adding a standby state to the payload Stateflow diagram impacted the results of the payload power draw. The results of the payload power analysis encouraged another look at the satellite's attitude which uncovered a required sun-soaking attitude phase based on the Grissom-2 mission power needs.

4.1 Orbital Analysis

There are many components of the satellite's mission that are considered in this research. The main components of the orbit that are analyzed consist of the semimajor axis and inclination. Due to the propagation of the satellite over time, the semimajor axis changes throughout the mission. In this scenario, the ideal altitude for most desirable payload data collection is at 450 km. To keep the satellite nearest to the ideal altitude, thrusting maneuvers were implemented as the satellite propagated toward the Earth. Initially, the satellite would thrust whenever the altitude was lower than 400 km, raising the orbit back to 450 km. Orbit raising of 50 km was possible for STK to implement, but did not prove realistic for the type of propulsion system being used. When orbit raising of 50 km was required, the delta V burns required were much greater than was reasonable for the cold gas thruster under investigation. With the capabilities of the thruster in mind, the allowable orbit degradation was changed to 20 km, instead of 50 km.

The burns required for a 20 km orbit raising were much closer to the actual capacity of the propulsion system. Of note, in both scenarios all payloads were turned off (not standby) to avoid hitting a fault mode, and instead focusing on the capabilities of the propulsion system. The 50 km scenario still hit a fault mode due to the solar array power generation dip, even before a thrusting maneuver would be able to take place. Because STK does not know about the power capabilities of the satellite, the scenario still ran—it is only in the post-processing that takes place in Simulink that the fault modes are accounted for. This is one example of how the PAT can be used to uncover system design issues unknown to the STK physics-based portion of the analysis. Table 9 below shows a comparison between the two sizes of orbit raising that were analyzed.

Table 9 - Orbit-Raising Comparison

Orbit-raising	**Raising Delta V (m/s)**	**Raising Burn Time (EpSec)**	**Circularizing Delta V (m/s)**	**Circularizing Burn Time (EpSec)**	**Total Fuel Consumed (kg)**	**Scenario Duration (EpSec)**	**Duration (Days)**
50 km	4.894729	5090100	14.129963	5109720	0.473	11418092.743	132.154
20 km	1.668560	2351280	5.915069	2370840	0.191	4104574.124	47.507

Creating a scenario with an ideal altitude of 450 km, means that using a goal of orbit raising whenever the altitude dropped below 430 km requires thrusting much more often than orbit raising at an altitude threshold of 400 km. If the satellite was launched into a higher orbit (less drag) with a greater semimajor axis, the satellite orbit would not degrade as quickly, and therefore may not need to perform orbit raising maneuvers as often. As seen in Table 11, the scenario time for the 20 km orbit raising was over 85 days (7.3×10^6 seconds) less than the 50 km orbit raising. The fuel consumed was also less at the 20 km orbit raising because of the lower delta V necessary but shorter mission duration. Depending on the thruster being used, the total delta V required for each maneuver must be evaluated. The user must understand what the

actual capabilities of the thruster are, and if the delta V results collected are consistent with what the actual thruster can produce. For example, just because STK allows for a high delta V to complete a maneuver, does not mean it is realistic for the propulsion system to produce that amount of delta V. Testing of the propulsion system before the satellite launches will give the user a better idea of the actual capabilities of the thruster and if the simulated scenario is possible. Further study of propulsion system capabilities is seen below in section 4.3.

A second mission component that is largely dependent on which orbit the satellite is launched into is the inclination. The inclination affects what geographic areas the payloads have access to while the satellite is orbiting around the Earth. The targets of concern for this mission scenario were at the following locations as seen in Table 10.

Table 10 - Target Locations

Target Name	**Latitude**	**Longitude**
AFIT Ground Station	39.782	-84.0832
Target 1 - Riyadh	24.7136	46.6753
Target 2 - Barcelos	-0.97357	-62.9269

Target 1 was of primary interest to Payload 1, Target 2 was of primary interest to Payload 2, and the third location is the AFIT ground station where all satellite data is downloaded. The location of the targets matters significantly in terms of amount of data collected and how often the data can be downloaded by the ground station. This mission set did not have issues with memory space available but changing the location of the ground station would either exacerbate or relieve a memory space issue. All MC3 network ground stations are present in the MatLab code and can be switched out for the AFIT ground station as necessary during specific mission analysis.

4.2 Quasi-static Mission Parameters

As described in section 3.1, the propulsion scenario was created to represent a quasi-static scenario. The goal is to create a scenario that aligns more with the reality of a satellite mission: the satellite propagates down to a degraded orbit and needs to thrust to raise the satellite back to its intended orbit, followed by another propagation. Each of the scripts builds on the previous one to determine when the propulsion system should thrust to best meet the needs of the mission. The first script produces a time when the first burn should occur to raise the altitude. That time is then used in the second script to execute the first burn. The second script produces a time when the second burn should occur to circularize the orbit. The times from the first and second script are used in the third script to execute both burns, followed by propagation back to a degraded altitude. The scripts are compounded to collect accurate information from the Simulink and Stateflow models that encompasses the full duration of the scenario. The following sections will describe the specifics of each part of the scenario and how the results can be modified to fit specific mission needs.

4.2.1 Integration Script 1

The first integration script propagates the satellite until it reaches 425 km and then ends the simulation. The simulation is run past the desired altitude of 430 km, to determine if there is a better altitude past 430 km to thrust at. After the data is collected through the various reports and is run through the Simulink and Stateflow models, a large amount of data is available for further analysis. Specifically, the data of interest to scenario part 1 is the time, battery DoD, satellite altitude, and access data for each target and the ground station. The data of interest are then put into a table within MatLab. If other data properties are of interest, they can be called from the telemetry data within Simulink and added to this table. The table is then parsed to find

the ideal time in the scenario where the satellite should stop propagating and thrust back into the desired orbit. The Stateflow diagram was the inspiration for creating the guards necessary to parse the data table to ensure that all parts of the simulation are consistent.

The time variable is what keeps the other parameters consistently associated with one another. Table 11 below shows the variables of interest and why they were chosen.

Table 11 - Integration Script 1 Parameters

Variable	**Guard**	**Reasoning**
Time	N/A	Any time after the scenario begins is an acceptable time.
Battery DoD	< 20%	A fault mode is triggered at a battery DoD greater than 30% and will not return to normal operations until it is lower than 20%.
Altitude	< 430 km	The altitude has dropped by 20 km, and now should return to its original orbit.
Target 1 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Target 2 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Ground Station Access	No access.	The payload mission takes priority, therefore in order to thrust, the satellite cannot be contacting a ground station to download payload data.

The script parses the data into a new table that only includes times where all of the above data is true. The user can then determine out of the suggested times, when the best time to burn may be.

For the sake of the simulation, the first available time was chosen to move on to the next part of the scenario. Table 12 is a sample of the suggested times as provided by the table. There was a total of 64432 different times that met the above conditions.

Table 12 – Integration Script 1 Results

	Time (s)	DoD (%)	Altitude (km)	Access Ground Station	Access Target 1	Access Target 2
1	2351280	3.0470	429.9846	0	0	0
2	2351340	3.1366	429.9614	0	0	0
3	2356860	3.0171	429.9055	0	0	0
4	2356920	3.1067	429.8591	0	0	0

4.2.2 Integration Script 2

The second part of the scenario includes the initial propagation, the first burn, and the transfer ellipse. The result of the second scenario is determining a time when the second burn should occur. The second burn circularizes the orbit, returning the eccentricity to zero. The transfer ellipse is run for 20,000 seconds. This enables the satellite to orbit the Earth 3.5 times to determine if there is a point that the satellite operations are favorable to burn at apoapsis, the most fuel-efficient time to burn.

Table 13 - Integration Script 2 Parameters

Variable	**Guard**	**Reasoning**
Time	>first burn	The second burn can only occur after the first burn.
Battery DoD	< 20%	A fault mode is triggered at a battery DoD greater than 30% and will not return to normal operations until it is lower than 20%.
Altitude	Apoapsis	The ideal placement of the second burn is at apoapsis. If the burn does not take place at apoapsis, it will take significantly more fuel to circularize the orbit.
Target 1 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Target 2 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Ground Station Access	No access.	The payload mission takes priority, therefore in order to thrust, the satellite cannot be contacting a ground station to download payload data.

After the data is parsed, the user needs to determine if there is a time at apoapsis where the above parameters are met. The time step for the data is set to 60 seconds, so there is likely not going to be a time available where the specific apoapsis altitude is present, as it may occur within the 60 second time step. For this research, the time closest to apoapsis was used to determine when to complete the second burn. To test if the burn is close enough to apoapsis to be most fuel efficient, a second simulation can be completed where the transfer ellipse ends at apoapsis

instead of at the time specified in integration script 2. If the burns are close enough in value, due to the specifications of the mission, it may be of more value to burn at the recommended duration than at the exact apoapsis value.

4.2.3 Integration Script 3

The third part of the scenario consists of one entire propulsion sequence for the mission. The simulation starts with a propagation, moves to the first burn, the transfer ellipse, the second burn, and finally an ending propagation before a second propulsion sequence is required. The script builds on the dynamic simulation created in the two previous scripts. Table 14 below explains the variables of interest for the third dynamic scenario.

Table 14 - Integration Script 3 Parameters

Variable	**Guard**	**Reasoning**
Time	>second burn	The next burn sequence can only occur after the second burn.
Battery DoD	< 20%	A fault mode is triggered at a battery DoD greater than 30% and will not return to normal operations until it is lower than 20%.
Altitude	<430 km	The altitude has dropped by 20 km, and now should return to its original orbit.
Target 1 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Target 2 Access	No access.	The payload mission takes priority, therefore in order to thrust, the payload cannot be accessing a target.
Ground Station Access	No access.	The payload mission takes priority, therefore in order to thrust, the satellite cannot be contacting a ground station to download payload data.

The result of the scenario tells the user when the satellite has propagated to an appropriate altitude and the next propulsion sequence should begin. The three scenarios can then be rerun to perform a second analysis. The information collected in these three scripts does not tell the user about the mission lifetime but does point to how often the satellite may need to thrust and on average how much fuel may be required. To gain a better understanding of the lifetime of the satellite, a series of these simulations would need to be run. Of note, the three scripts must build upon each other to retain accurate results, meaning that if parameters are

changed in the first script, they also must be changed in the second and third scripts. Using CATIA helps alleviate this issue because there is one place where all parameters are defined within the value properties of the model, as seen in Figure 23. The accuracy of the results collected depends on the amount of information available for the components being tested. The more value properties present for each component of the satellite or the mission, the more detailed information STK has available to run an accurate simulation. When STK first creates a scenario, there are many defaults that are inherently created around the scenario. As those defaults are switched out to represent the mission at hand, the fidelity of the scenario is increased, and therefore the information collected from the analysis is more accurate to the mission under investigation.

4.3 Propulsion Analysis

Using the scenario described in section 4.3, the following propulsion trade study was conducted. As described above in section 3.5 the main parts of the propulsion subsystem that are of interest are the type of thruster, Isp (s), thrust (mN), number of thrusters, thrusting power, and fuel mass. When considering working with COTS components, these variables are typically defined based on the type of thruster used. In this research, an impulsive burning cold gas thruster was of interest. Table 7 – STK Engine Parameters, define the parameters of the simulated COTS propulsion subsystem, which largely affect the delta V able to be produced by the propulsion system and the fuel required to complete the maneuver. When running the mission in STK, the simulation does not take into account the actual capabilities of the propulsion system, but rather runs the scenario with the information provided. The simulated thrusters examined in this analysis are seen in section 3.5. Thruster #1 contains the parameters

that were used throughout the scenario as the standard propulsion system. The other thrusters were only used for this propulsion trade study and not in any further analysis.

The analysis below is appropriate for a situation where there is a fixed mission, but an option to trade components to best fit the needs of the mission. Propulsion subsystem attributes of main concern in this analysis are fuel used and delta V capacity. As mentioned above, STK will produce the scenario based on the attributes provided, not necessarily based on the actual feasibility of the simulation propulsion system. To find a thruster that most matches the needs of the mission, all of the thrusters were run through the scenario with an orbit raising of 20 km and produced the following results in Table 15. Throughout each scenario, all payloads were turned off to avoid any fault modes prior to the burns.

Table 15 - Propulsion subsystem Trade Study Results

Thruster	**Raising Burn**		**Circularizing Burn**		**Total Fuel Consumed**	**% Fuel consumed**
	Delta V (m/s)	**Duration (s)**	**Delta V (m/s)**	**Duration (s)**		
#1	1.668560	166.502	5.915069	584.578	0.191	15.47%
#2	1.762807	119.179	N/A	N/A	0.046	N/A
#3	5.354400	441.981	6.719143	546.166	0.252	142%
#4	1.818442	85.859	5.881574	274.991	0.184	30.67%
#5	6.984548	1256.868	N/A	N/A	0.214	N/A

Thruster #1 is the standard thruster used throughout this research and is a baseline for the propulsion trade study. Thruster #2 completed the first burn with a reasonable delta V. The issue came after the first burn was completed due to the large amount of power required to heat the thruster (55 W). Approaching the second burn the satellite enters a Sun Safe fault mode and

cannot recover from it before entering a survival fault mode. Due to these faults, the second burn never occurs. Due to the large amount of power required to heat the thruster which causes a strain on the battery, this propulsion system is incompatible with the mission.

Thruster #3 is also unable to complete the mission. After the first burn, 64% of the total fuel is used. The second burn is unable to be completed without running out of fuel. If both burns were completed, the thruster would have used 142% of its fuel rendering this propulsion system incompatible with the mission.

Thruster #4 received similar results to Thruster #1, though using a greater percentage of total fuel meaning that it would be able to complete fewer orbit raising maneuvers than Thruster #1. Thruster #4 has double the number of thrusters which cuts the amount of time spent burning in half. The greater number of thrusters creates a more impulsive type burn, completing the first maneuver in less than 1.5 minutes and the second burn in just over 4.5 minutes. Further research into the differences between an impulsive burning thruster versus a continuous burning thruster may prove to find a propulsion system that meets the Grissom mission needs to an even greater extent and provide greater fidelity to the current model.

Thruster #5 had similar results to Thruster #2. After the first burn, the satellite entered a sun safe fault mode followed by a survival fault mode due to the large amount of power needed to heat the thruster (20 W). As the solar array power generation began to dip, the battery DoD could not sustain the power draw. Due to the large amount of power required for Thruster #5, it is deemed incompatible for this mission.

Thruster #1 and Thruster #4 were the only propulsion systems that successfully completed the mission within the given parameters. Similar to any other component, there were

specific limitations surrounding each propulsion subsystem, forcing the simulation into a fault mode, or in Thruster #3's case running out of fuel. This is just one example of a component trade study. Each component of the satellite can have a similar trade study to find which combination of components may meet the mission requirements. By using the PAT, these trade studies can be completed quickly and efficiently, without expending physical resources. Any of the variables can be manipulated in CATIA, creating an easy way to test different components without putting a strain on the physical component or the satellite team.

4.4 Payload Analysis

4.4.1 Payload State Analysis

As explained in section 3.4.1, payload testing primarily consisted of comparing the results of Brown's PAT, to the updated Simulink and Stateflow models using Erbe and Lemmer's updates. The payload testing was conducted using Brown's one-week scenario. A comparison of the data collected can be seen below in Figures 35-36

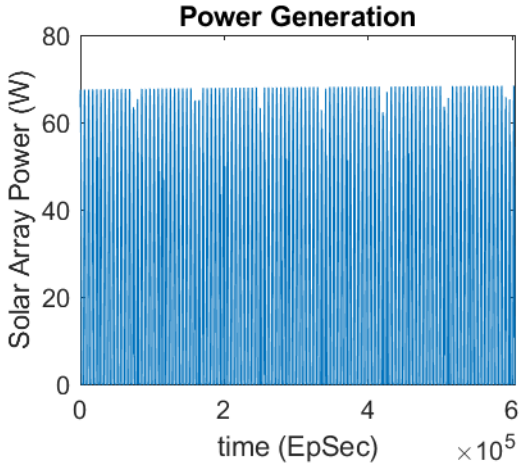

Figure 35 – Brown's Solar Array Power Generation

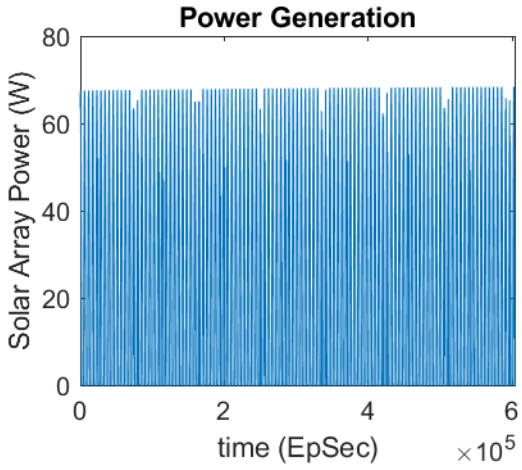

Figure 36 - Erbe and Lemmer Solar Array Power Generation

The solar array power generation remained the same between both trials. This means that the same amount of power was available throughout the lifetime of the scenario. When comparing the power levels of the payloads, it is important to understand how much power was available to the battery, especially in the case that a battery DoD fault is triggered.

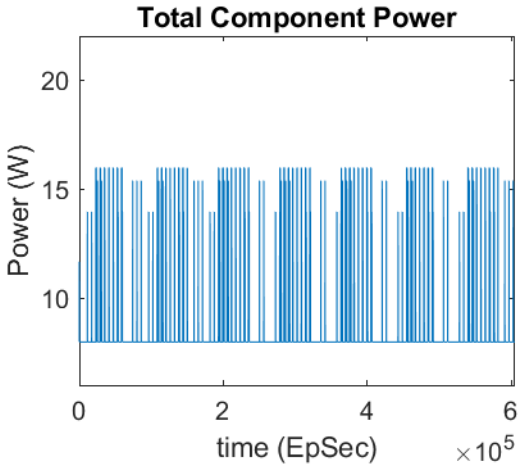

Figure 37 – Brown's Total Component Power

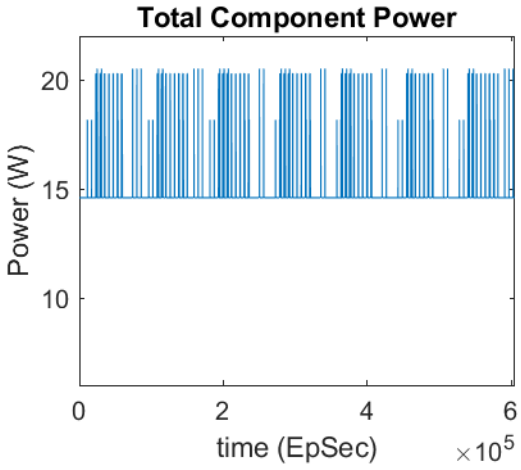

Figure 38 - Erbe and Lemmer Total Component Power

The total component power is the amount of power necessary to operate the various components of the satellite throughout the duration of the scenario. The total component power saw the most change between payload values upon the Erbe and Lemmer updates. The total component power increased from Brown's PAT to Erbe and Lemmer's updated PAT as seen in Table 16.

Table 16 - Min and Max Total Component Power Comparison

PAT Version	Minimum Total Component Power (Watts)	Maximum Total Component Power (Watts)
Brown	7.9850	15.9850
Erbe and Lemmer	14.6090	20.5410

The greater power is due to the payload entering a standby state instead of an off state between target passes. When the payload is in a standby state, there is still a power draw from the payloads. Table 17 below shows a breakdown of the power levels and data rate levels of the payloads based on their state.

Table 17 - Payload Value Properties

Payload	Standby Power (Watts)	On Power (Watts)	Standby Data Rate (Bytes/sample time)	On Data Rate (Bytes/sample time)
TIC Imager – Payload #1	1.2	6.55	0	1916.67
Sensor SDR – Payload #2	2	5	0	1100
Beacon/Comms – Payload #3	1.5	4	250	500

Figures 39 and 40 show the combined payload power over the lifetime of the scenario for both Brown and Erbe and Lemmer's PATs.

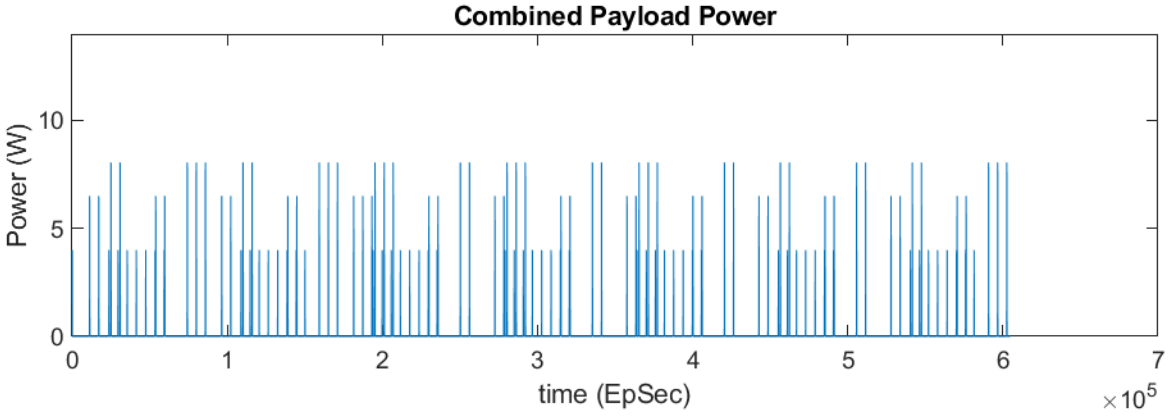

Figure 39 – Brown's Payload Power

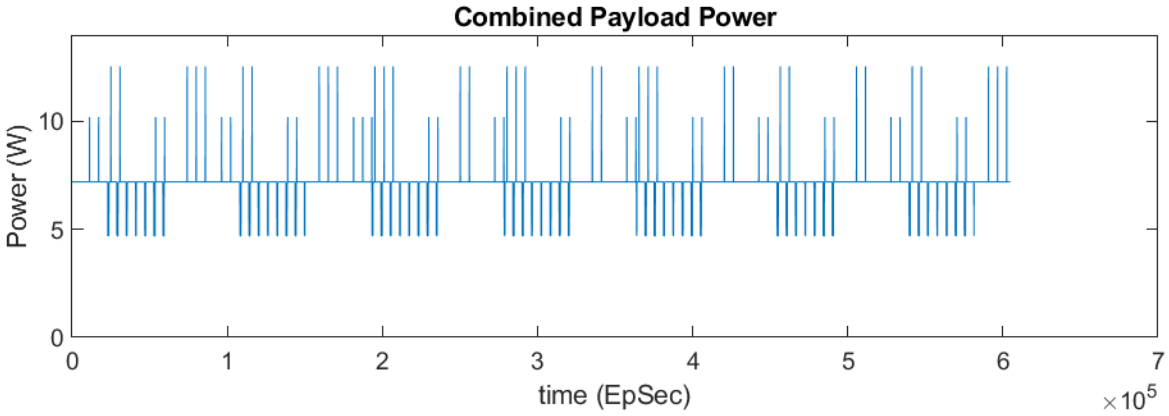

Figure 40 - Erbe and Lemmer Payload Power

Based on the payload state analysis above, including a standby payload state increases the combined payload power over the entire scenario. The ramifications of the state change are seen in the sections below. These tests brought clarity to the design limitations of the chosen payloads, as well as greater information concerning the solar array power generation.

4.4.2 Payload Power Analysis

The updates in the payload states and addition of a propulsion system proved to have the greatest increase in the total component power levels of the satellite. A series of scenarios were run where a combination of payloads and the standard thruster (Thruster #1) were examined, therefore producing variable power levels. The basic quasi-static scenario as described in section 4.3 was used when conducting this analysis. Table 18 below shows the separate combinations, the maximum and minimum total component power required for the mission, and whether a fault was triggered. Column 1 describes which trial is being run, while columns 2-4 describe if a payload is on or off for the trial. If the payload is on, it will draw power based on the states described in the Stateflow model. If the payload is off, it will not draw power in any state.

Table 18 - Variable Payload Power Analysis

Trial	Payload #1 On?	Payload #2 On?	Payload #3 On?	Max Total Component Power	Min Total Component Power	Fault Triggered?	Sun Safe Fault Time (1×10^6 EpSec)	Survival Fault Mode Time (1×10^6 EpSec)
1	No	No	No	19.9939	8.9850	No	N/A	N/A
2	Yes	Yes	Yes	21.5412	4.2000	Yes	0.9750	0.9962
3	Yes	No	No	18.0890	4.2000	Yes	1.758	1.764
4	No	Yes	No	18.8250	4.2000	Yes	1.624	1.630
5	No	No	Yes	18.3650	4.2000	Yes	1.366	1.372
6	No	Yes	Yes	20.2050	4.2000	Yes	1.113	1.120
7	Yes	No	Yes	19.5412	4.2000	Yes	1.200	1.235
8	Yes	Yes	No	19.9290	4.2000	Yes	1.445	1.450

The only scenario where a fault mode was not triggered was Trial 1 where all payloads were turned off. Trial 1 was the only scenario where the thrusting maneuvers were completed and shown in the Simulink results. All other trials hit a survival fault mode before the thrusting maneuver was necessary. The minimum total component power was highest for Trial 1, which seems backward because all payloads were turned off. When the satellite enters a fault mode, all component power turns to zero except for the T&C subsystem, C&DH subsystem, ADCS subsystem, and propulsion subsystem. The T&C receiver is always on (0.7 W), while the transmitter fluctuates between on and off depending on if it is contacting the ground station. The C&DH subsystem is always on (2.5 W). The propulsion subsystem remains in a standby mode when not heating or thrusting (1 W). When adding all of the power necessary for the three subsystems that remain active even in a fault mode, it equals the 4.2 W as seen in the minimum

total component power value. The ADCS subsystem turns off only when in survival mode but will continue to operate in sun safe mode. At any time the satellite is operating without a fault mode, it will always require at least 8.8950 W to operate. This minimum value is true with or without payloads. Due to all trials with operational payloads hitting a fault mode, there must be a change to the satellite components if the mission is to operate longer than two weeks. Below in Table 19 is the same set of trials, but the power required for the payloads in standby mode was set to zero watts. This essentially removes the standby mode for the payload and simply switches the payload between an on and off state.

Table 19 – Variable Payload Power Analysis with Standby Equal to Zero Watts

Trial	**Payload #1 On?**	**Payload #2 On?**	**Payload #3 On?**	**Max Total Component Power**	**Min Total Component Power**	**Fault Triggered?**	**Sun Safe Fault Time (1×10^6 EpSec)**	**Survival Fault Mode Time (1×10^6 EpSec)**
1	No	No	No	19.9939	8.9850	No	N/A	N/A
2	Yes	Yes	Yes	19.5412	4.2000	Yes	1.371	1.377
3	Yes	No	No	19.9939	8.9850	No	N/A	N/A
4	No	Yes	No	19.9939	8.9850	No	N/A	N/A
5	No	No	Yes	16.9850	4.2000	Yes	1.371	1.377
6	No	Yes	Yes	17.2650	4.2000	Yes	1.371	1.377
7	Yes	No	Yes	19.5412	4.2000	Yes	1.371	1.377
8	Yes	Yes	No	19.9939	8.9850	No	N/A	N/A

Table 19 is one example of a change made to the satellite components which extends the life of the mission and capabilities of the satellite. Whenever the simulation ran to completion and did

not enter a fault mode, Payload #3 was turned off with a minimum total component power of 8.9850 Watts and a maximum total component power of 19.9939 Watts. In the trials that entered a fault mode, Payload #3 was turned on and faulted at the same time -- entering Sun Safe mode at 1.371 $x10^6$ EpSec. In Trials 2, 5, 6, and 7 in Table 17 where the standby mode was drawing its full allotted power, the Sun Safe fault mode was entered before 1.371 $x10^6$ EpSec each time that it faulted. This makes sense because more power was being used in Table 18 than in Table 19 because of the additional standby power. As the solar array generation begins to dip, the satellite's battery cannot withstand the power draw necessary to operate all of the payloads at full power. Trials 3, 4, and 8 in Table 19 successfully completed the scenario without a fault, where previously in Table 18 all of those trials entered a fault mode. By changing the power draw of the payloads' standby modes, the battery's DoD remained stable enough to continue the mission. Even when Payload #3 changed the standby power to zero Watts, the missions with Payload #3 were never completed. This concludes that Payload #3 is incompatible with the Grissom bus and should not be used for the mission. Payload #3 is almost always on while operating in normal conditions. While the satellite is sun soaking, the payload is on. While the satellite is in bus operations mode but not sun soaking, either contacting the ground station or thrusting, the payload is in standby mode. Payload #3's incompatibility is just one example for how the PAT can be used to analyze a mission and component synchronization.

4.4.3 Solar Array Power Generation

Other than total component power changes based on the scenario updates, there were also changes in the solar array power generation. As the scenario time was increased from one week in Brown's scenario to seven weeks in the updated propulsion scenario, the solar array power generation changes were more apparent. The solar array power generation largely affects the

battery DoD and if the satellite will enter a fault mode due to lack of power. Below are a series of solar array power generation plots, based on a scenario of one week, one month, and three months. The satellite is in STK's default satellite attitude, nadir alignment with ECI velocity constraint. The nadir alignment has the +Z-axis aligned with nadir and the X-axis constrained in the direction of the velocity vector. For reference, the Grissom-2 solar arrays are in the -Z-axis. The scenario is a simple satellite propagation without propulsion, meaning that as the scenario unfolds, the satellite orbit degrades without orbit raising. Though a different scenario would produce slight changes to the solar array power, having the satellite propagate is essential to show the power levels over the lifetime of a satellite.

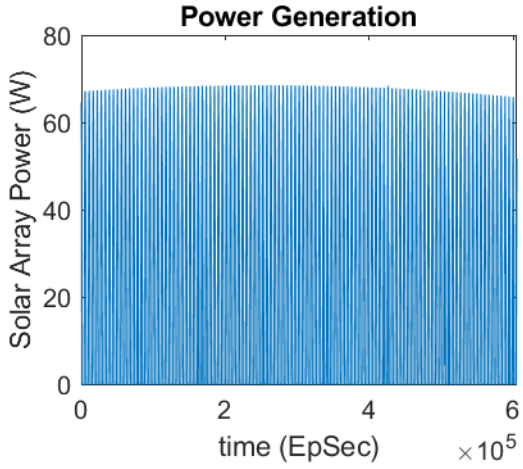

Figure 41 - 1-week Solar Array Power Generation

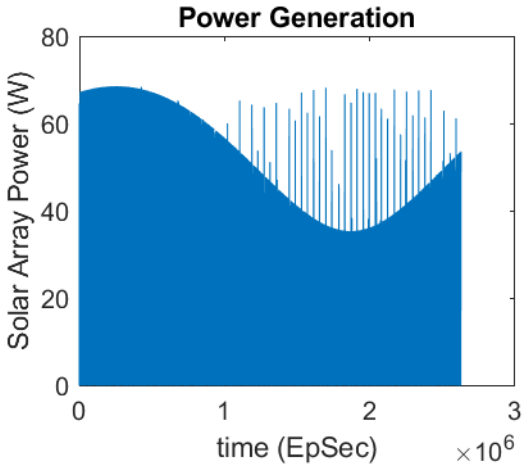

Figure 42 - 1-month Solar Array Power Generation

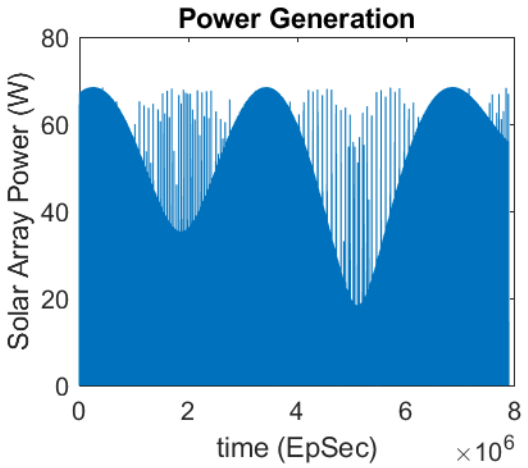

Figure 43 - 3-month Solar Array Power Generation

The larger dips in power as seen around $2x10^6$ and $5x10^6$ seconds, are typical areas where a battery DoD fault may occur, depending on how much power draw is occurring from the satellite. Brown's scenario only ran for one week, not giving the fullest picture of what would happen to the power generation over the entire satellite mission. Initially it was thought that these dips are what cause the battery DoD faults, but upon further research it was not the dips that cause the battery DoD faults but the duration the satellite spent in a directly sun-soaking state. To alleviate this issue, changes were made to the attitude of the satellite.

The default satellite attitude in STK is Nadir alignment with ECI velocity constraint. The Grissom-2 mission has the solar panels in the -Z direction, so with the +Z axis pointed nadir in the default attitude, it restricts direct sun-soaking but does not completely isolate the satellite from the sun. Based on the research mentioned above, depending on how much power the payloads are drawing, the satellite may perform very well in this attitude. Udell's research looked into changing the attitude of the satellite to achieve a sun-soaking state in the -Z direction [33]. She found that using STK's XPOP Inertial attitude allowed for sufficient sun soaking of the satellite. XPOP orients the satellite so that the X-axis is normal to the orbit plane while the Z-axis and Y-axis remain within the orbit plane. The -Z vector is always in alignment with the sun vector [33]. By using the XPOP attitude for the propagation simulation, the solar array power generation results (Figures 44-46) proved to have a similar pattern to the results experienced using the nadir with ECI constraint attitude used previously (Figures 41-43).

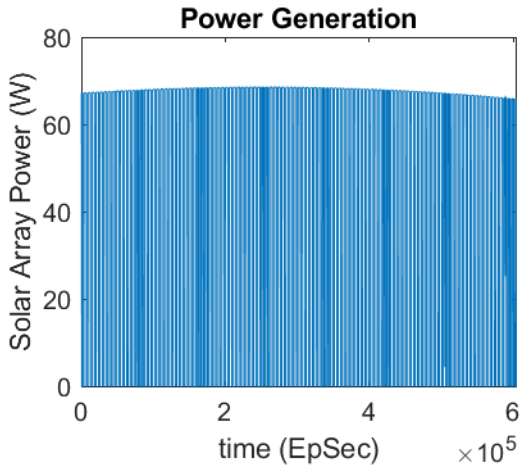

Figure 44 - 1-week XPOP Solar Array Power Generation

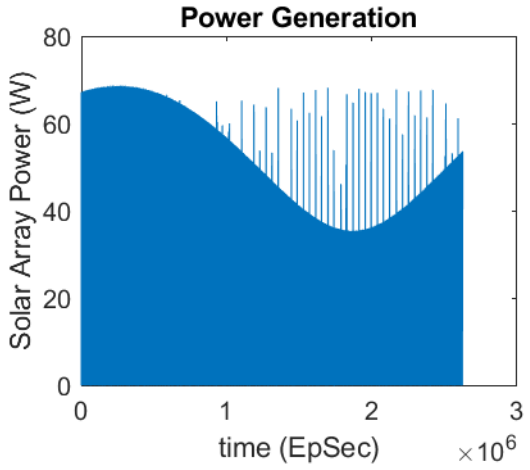

Figure 45 - 1-month XPOP Solar Array Power Generation

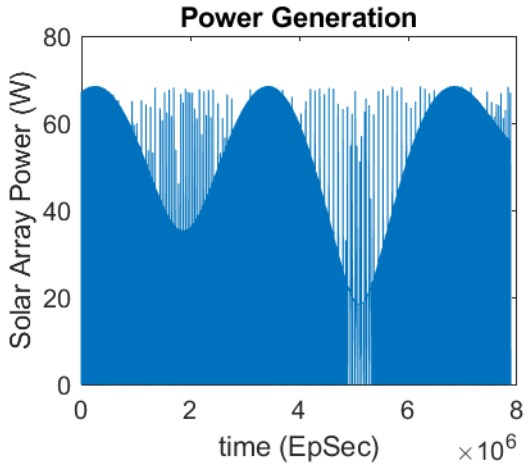

Figure 46 - 3-month XPOP Solar Array Power Generation

Initially, these similar results were concerning. Even with all payloads at full power, the battery DoD never was high enough to enter a fault mode of any kind using the XPOP attitude, but how could that be with such similar power dips? Further investigation was made into the Simulink plots of the XPOP solar array power generation data. Figure 47 shows the solar array power generation data over 1-month for XPOP and figure 48 over 1-month for the default attitude.

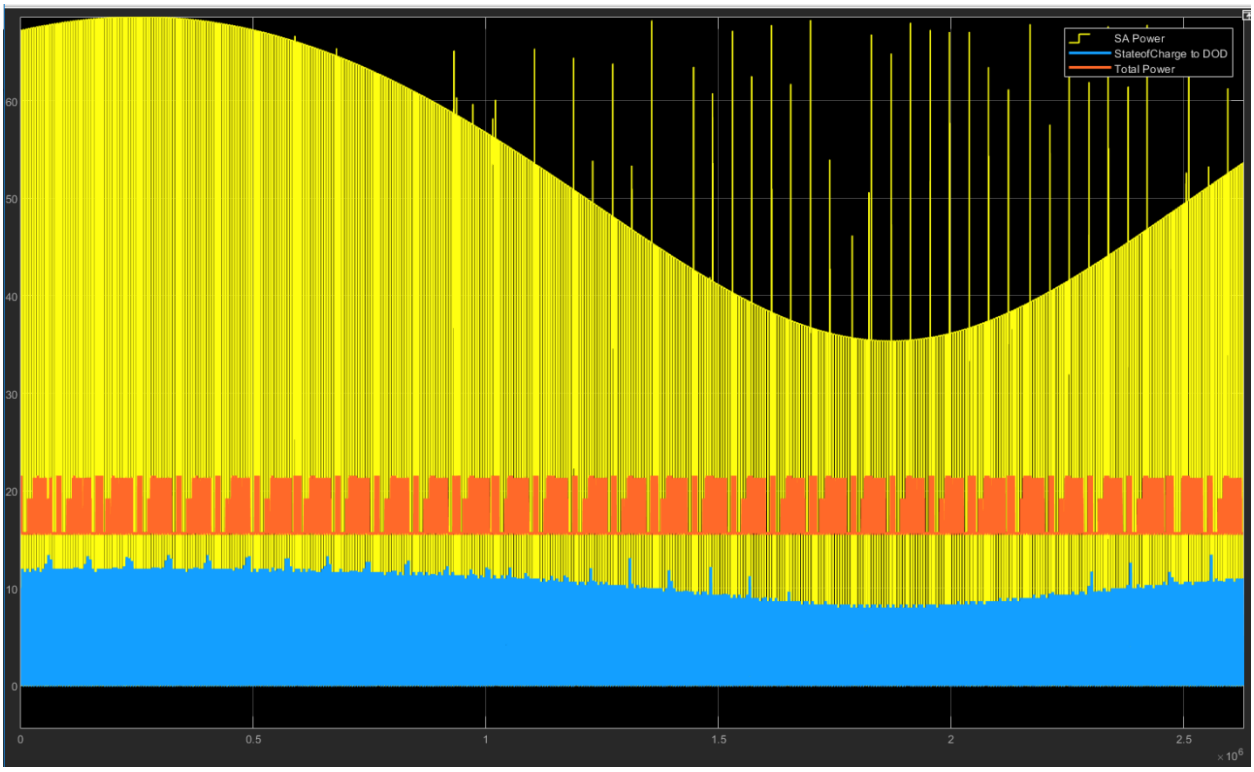

Figure 47 - XPOP Solar Array Simulink Data

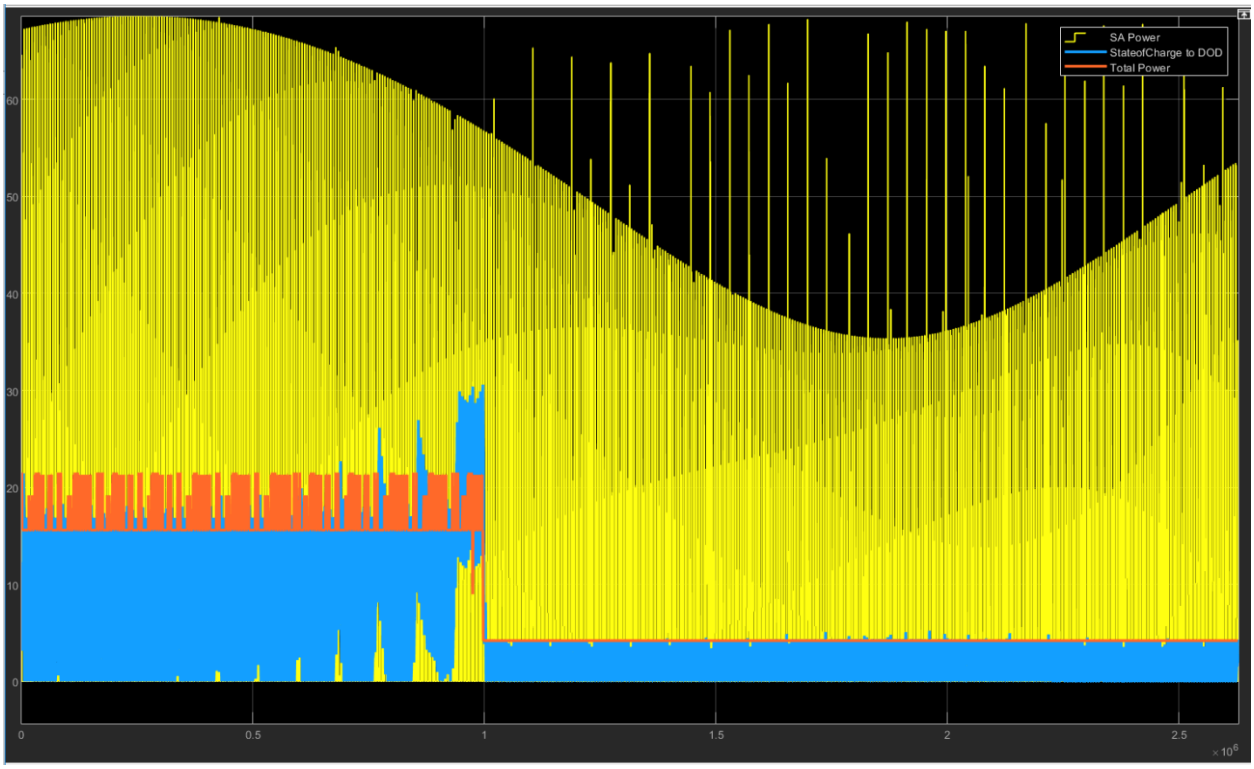

Figure 48 - Nadir with ECI constraint Solar Array Simulink Data

The yellow represents the solar array power generation, the orange represents the total component power, and the blue represents the battery DoD. Both Figures 47 and 48 have a similar overall pattern of solar array power generation, but Figure 47 has a much more consistent transition from sun-soaking to eclipse. Figure 48 shows more variability with when the satellite is sun-soaking and entering into an eclipse.

Taking a further look into the Simulink data, the data was parsed into 1-day simulations with the results below in Figures 49 and 50.

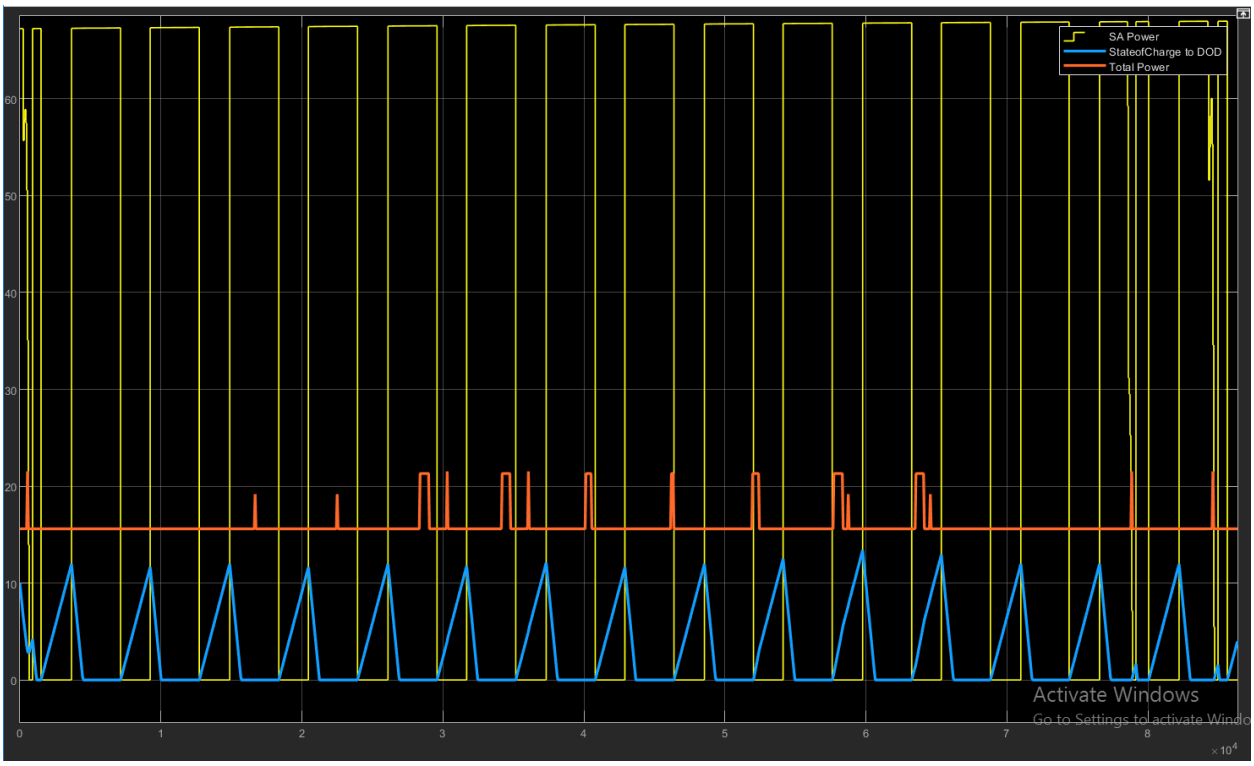

Figure 49 - 1-day XPOP Solar Array Simulink Data

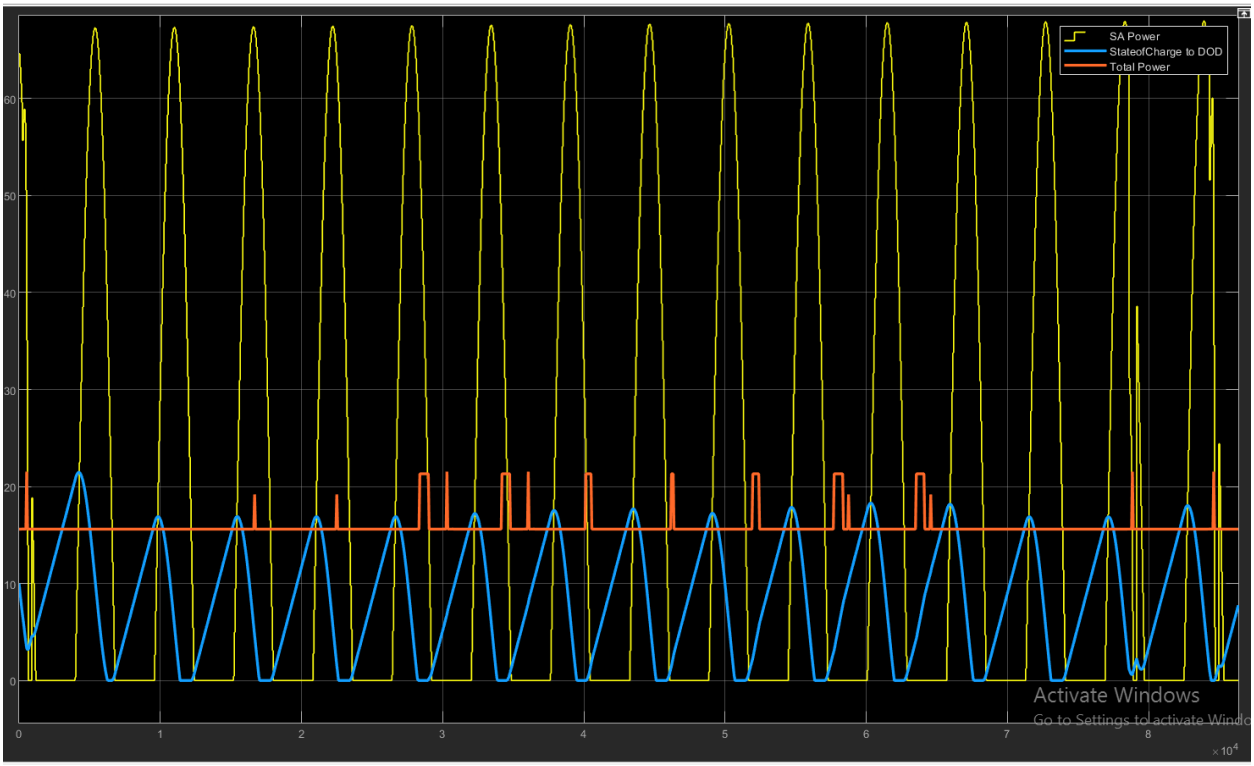

Figure 50 - 1-day Nadir with ECI constraint Solar Array Simulink Data

Figure 50 shows the solar array achieving the max power generation, but for only a brief period of time before the power generation drops back down during eclipse. The battery DoD is consistently greater in Figure 50 than Figure 49. In Figure 49, the solar arrays spend a greater amount of time at the maximum solar array power generation than in Figure 50. In Figure 49, the solar arrays spend more time at the maximum solar array power generation than in eclipse. The longer stretches of maximum power generation using XPOP allow the battery to have a greater extended charge when compared to the default attitude.

4.5 Summary

In summary, Chapter IV illuminated the results of the propulsion scenario, to include the propulsion trade study and an in-depth look into the changes in payload states. Thruster #1 was found to be the propulsion system best suited for the Grissom-2 mission due to its avoidance of fault modes and low percentage of fuel consumed. By completing a series of tests involving the payload states, it was revealed that in the default STK satellite attitude, payload-3 is incompatible with the Grissom-2 mission. To alleviate the large power draw of payload-3, changes may be made to the satellite's attitude. By changing the attitude from the default to the XPOP attitude, the solar arrays were able to stay in a higher power producing sun-pointing state over longer periods of time, avoiding faults generated by a high battery DoD.

V. Conclusions

5.1 Summary of Research Questions

All research questions discussed in Chapter I were sufficiently examined and answered. A summary of the questions and their results are found below.

How can a propulsion system be integrated into a model to validate payload mission success and to what fidelity? A propulsion subsystem was added to all aspects of the model to include CATIA, STK, and Simulink/Stateflow. The propulsion subsystem was used to model an orbit-raising scenario within STK, where the resulting data was then input into the updated Simulink and Stateflow models to produce the necessary analysis. The propulsion subsystem was analyzed alongside the payloads to ensure that the mission could be completed with adequate power. The fidelity of the propulsion system being used in the mission can be determined based on the post-analysis of the PAT results. When conducting the propulsion trade study, the values of delta V were examined to determine if the values STK produced, were truly what the propulsion system could handle.

What figures of merit should be considered for payload mission analysis when considering CubeSat propulsion with varying payloads? The figures of merit taken into account for the payload mission analysis when considering propulsion were two-fold. The parameters specific to the propulsion subsystem included delta V and fuel consumed, while the parameters for the entire mission were total component power, and faults triggered.

What are the parameters necessary to evaluate the propulsion system alternatives for a CubeSat mission? The propulsion system parameters necessary for evaluating the propulsion alternatives are Isp, thrusting magnitude, number of thrusters, fuel mass, dry mass, standby power, heating power, and thrusting power. All of the parameters are value properties within the

model and are necessary for creating a propulsion scenario with a custom engine in STK. If available, addition parameters can be added to increase the fidelity of the model, but the parameters listed above are the minimum necessary to create an accurate custom engine.

5.2 Contributions

This research effort began with Brown's PAT, which extended the LMMT simulations to include multiple payloads and utilize current MBSE tools to further integrate the usefulness of analyzing multiple payloads on one CubeSat mission. Brown's scenario was static and only lasted one-week, giving a small sample of what the lifetime of a CubeSat mission may look like. This research modified Brown's PAT to include Erbe and Lemmer's updated payload state findings, adding a propulsion subsystem, and creating a quasi-static scenario that only utilized the propulsion subsystem if all necessary mission parameters were met.

By adding Erbe and Lemmer's updated payload states including a standby power state, the power levels of the payloads were significantly altered. This revealed a potential lack of power throughout the lifetime of the mission depending on the standby power draw of the individual payloads. Updating the payload states made it apparent that if there are errors in the state progressions of any of the components, the results of the PAT may be changed significantly.

A larger contribution of this research was adding a propulsion subsystem to the model. The propulsion subsystem was added throughout the model, to include CATIA, STK, and Simulink. Adding the propulsion subsystem allowed for the simulation of an orbit-raising scenario to be proven and analyzed, requiring significant updates to the MatLab script that directs the STK simulation. The updates to the CATIA model were implemented using the pre-programmed propulsion subsystem already present in the CRA. The updates made to Simulink

and Stateflow added an additional Thrusting state to account for the maneuvering. A propulsion trade study was conducted to test the model and find a propulsion subsystem that would best meet the requirements of the mission. The trade study revealed multiple incompatible propulsion systems and showed how simple conducting a trade study using the PAT is for the user. The propulsion trade study provided an example for how other component trade studies may be conducted by the satellite user. Adding the propulsion subsystem to the model required the model to be changed from static to quasi-static. The changing parameters of the CubeSat components and payload mission required specific timing to be taken into account when finding the optimal time to thrust. The quasi-static model can easily be updated to suit mission needs and altered to take into account other areas of interest in the scenario. Depending on the limitations of the satellite program, much of the time it is either the components of the subsystems, or the mission itself that must be changed in order for a successful mission to take place. With CubeSats oftentimes being a part of a rideshare mission, this means the CubeSat may have little control over the orbital components of the mission, making it even more necessary to understand how subsystem components may be altered to ensure mission success.

Creating a PAT with an extended scenario time, to include the use of propulsion, allows the satellite user to uncover issues over the lifetime of the mission that may not be apparent in a shorter simulation. The payload power issues found after integrating Erbe and Lemmer's payload state updates, as well as the solar array generation dips, were not evident in previous versions of the PAT and forced the satellite to enter unexpected fault modes. The orbit-raising tests showed that even though the simulation was successfully executed in STK, the delta Vs required to complete the maneuver may not be feasible for the selected thruster, therefore necessitating a

shorter orbit-raising scenario that requires less thrust or changing the propulsion system out for a more powerful thruster.

5.3 Limitations and Lessons Learned

5.3.1 Limitations

The mission scenario is hardcoded within MatLab and is therefore not easily altered from CATIA as the primary variable or parameter interface. The maneuvers within the simulation are set in place, though specific attributes of the maneuvers can be easily changed from CATIA. The simulation only accomplishes a single orbit-raising scenario. In order to accomplish multiple burns to fulfill the simulated life of the satellite, additional integration scripts would need to be created and added to the scenario.

The scenario is created for an impulsive thrust propulsion system. If a continuous/finite thrust propulsion system is to be used in this scenario, code within the two thrusting maneuvers in the MatLab script will need to be altered. STK provides a number of resources to explain how to change the code to accommodate a finite thrust propulsion system.

Within Simulink and Stateflow, the diagrams are created specifically for the components being tested. If a trade study is necessary where the mechanisms of the component are drastically different or alternate forms of data are needed from STK in order to fully validate the component, those changes will need to be made within the Simulink and Stateflow models. Adding an additional component such as the propulsion system did not prove to be as difficult as editing the existing mathematical components within Simulink.

5.3.2 Lessons Learned

The interface between CATIA and MatLab can prove problematic. The error that most frequently occurs within CATIA when calling MatLab through the PAT includes the following:

"ERROR: The MATLAB Engine may be closed. Please execute "kill matlab" to restart the MATLAB engine via console."

This error may occur multiple times. Enter *kill matlab* into the console and try to run the parametric diagram again. If problems continue to occur, or the simulation is not loading, terminate the execution and start again.

When editing the scenario beyond simply changing the value properties within CATIA, it is much easier to run the scenario directly from MatLab. To run from MatLab, un-comment all of the variables at the top portion of the code and run the script. MatLab will call the Simulink diagram and produce all plots specified by *makefigures.m*. This is an easy way to check if your code is working properly without also having to troubleshoot interface issues with CATIA.

If the STK mission itself needs to be altered outside of simply changing the value properties, the most streamlined approach is using STK's connect command library. If the connect command library does not have the specific attribute you are looking for, try to create the desired scenario using the STK GUI. By using the GUI, you can begin to understand the steps needed to write the code and where in STK the necessary parameters need to be edited. After you have created the scenario with the GUI you will have to go back and script the code in MatLab, but at least you have a better idea of where to begin.

5.4 Areas of Future Work

There are multiple areas where the PAT could be further expanded and created to produce a mission scenario that is most closely related to reality. The propulsion trade study can be expanded to include other types of thrusters. Currently, only cold gas thrusters were used in the analysis, though cold gas thrusters are not the only type of thruster that has been tested on a CubeSat. Electrospray thrusters are the other type that have successfully been implemented on a CubeSat. Doing a trade study between different thruster types may be of interest in finding which propulsion system ultimately would be best to meet mission requirements.

Aside from propulsion systems, trade studies for other CubeSat components would also be of interest. Due to the nature of CubeSats, there are many components that can be switched out or modified with other COTS components. Doing a trade study on other CubeSat components such as the solar arrays, ADCS, or battery, may prove to find a component solution that both meets mission requirements and performs better than expected. Performance trade studies make room for even further mission discussions concerning cost and schedule. If the PAT can show that one component outperforms another on this specific mission, there may be incentive to trade cost or schedule in other areas of the project to include the proven component.

As the model currently stands, once the satellite enters the survival fault mode, there is no way for it to exit. Depending on the mission and capabilities of the satellite, exiting survival mode may be of interest, especially when considering the large fluctuations in solar array power generation throughout the mission.

Another area of further work involves the automation of the scenario. Instead of including three separate scripts for one propulsion maneuver, automating the scenario through the use of one script. Automation allows for less user error due to manually inputting

information, and the ability to use one interface in controlling the model. Automating the scenario may also allow for an ability to loop the scenario for the lifetime of the mission. Instead of producing one orbit raising maneuver, the simulation may produce multiple throughout the intended life of the mission. By enabling automation to run a longer scenario, further issues may be revealed, similar to the solar array power generation issue mentioned earlier.

5.5 Summary

This research with the updated PAT demonstrates the benefits of simulating a propulsive satellite mission with a dynamic mission scenario. The model is highly modular due to the usefulness of the MBSE tools within the CRA, giving CubeSat users the ability to conduct efficient trade studies with a propulsion system already available as an example. By extending the length of the scenario, the user has a better idea of constraints over the mission lifetime the satellite may experience. This model will serve AFIT in the immediate future as it prepares to choose subsystem components for the upcoming Grissom-2 mission and leads the way for digital engineering within the Air Force and Space Force.

Appendix A. PAT Value Properties

The following value properties were used in the Payload Analysis Tool. These were the value properties used throughout the simulations unless otherwise stated in the document. At times in the model, the value properties have a different name than the port. The port names are what are used as the variable within the MatLab script and should not be changed. The name of the value properties may be changed without corrupting the MatLab script.

Table 20 - PAT Value Properties

Value Property Name	Port Name	Value
STK Scenario Parameters		
dynamicPropStop1	dynamicPropStop1	2351280 seconds
dynamicPropStop2	dynamicPropStop2	2370840 seconds
stkVersion	stkVersion	12
scenarioName	scenarioName	GrissomMission
scenarioStartTime	scenarioStartTime	1 Mar 2023 12:00:00.000
scenarioStopTime	scenarioStopTime	+1year
deorbitAltitude	deOrbitAltitude	50.0 km
stoppingAltitude	stoppingAltitude	425.0 km
Orbit		
semiMajorAxis	semiMajorAxis	6828.137 km
eccentricity	Eccemntricity	0 degrees
inclination	Inclination	60 degrees
argumentOfPeriapsis	argumentPeriapsis	0 degrees
RAAN	RAAN	0 degrees
trueAnomaly	trueAnamoly	0 degrees
Cd	Cd	2.2
AFIT MC3 Remote Ground Terminal		
facName	facName	MC3 AFIT
facLatitude	facLatitude	39.782 degrees
facLongitude	facLongitude	-84.0832 degrees
CubeSat		
satelliteName	satelliteName	Grissom-2
SRPArea	SRPArea	0.2 m^2
dragArea	dragArea	0.2 m^2
totalMass	totalMass	10.0 kg
dryMass	dryMass	8.765 kg
radiationPressureArea	radiationPressureArea	0.2 m^2
Target1		

target1Name	Target1Name	Riyadh
target1Latitude	Target1Latitude	24.7136 degrees
target1Longitude	Target1Longitude	46.6753 degrees
Target2		
target2Name	Target2Name	Barcelos
target2Latitude	Target2Latitude	-0.9757 degrees
target2Longitude	Target2Longitude	-62.9269
C&DH		
busSOHDataRate	busSOHDataRate	9600.0 Bytes per second (Bps)
miscDataProductionRate	miscDataProductionRate	0 Bps
totalStorageCapacity	storageCapacity	8 GB
onPower	CDHPowerOn	2.5 W
initialMemoryFill	initialMemoryStorageStatus	1.0 $x10^9$ Bytes
CDHtotalMass		266 grams
Battery		
initialFill	batteryInitialFill	90 %
capacity	batteryCapacity	6 Ahr
nominalVoltage	battNominalVoltage	14.4 V
chargeDischargeEfficiency	chargeDischargeEfficiency	0.995
distributionEfficiency	distroEfficiency	0.9
maxDepthOfDischarge	DODtriggerSunSafe	30%
depthOfDischargeFaultRecovery	DODreturnToNormalOperations	20%
Solar Array		
PPTefficiency	PPTefficiency	0.95
area	solarArrayArea	0.1702 m^2
arrayEfficiency	solarArrayEfficiency	0.295
Reaction Wheel		
maxPower	reactionWheelMaxPower	1.925 W
zeroTorquePower	zeroTorquePower	0.27 W
maxMomentum	maxMomentum	0.015 kg*m/s
reactionWheelMomentOfInertia	reactionWheelMOI	0.0010577 Nm/s
initialRPM	initialRPM	0
reactionWheelMaxRPM	reactionWheelMaxRPM	5200
Magnetometer		
averagePower	magnetometerOnPower	0.825 W
torqueCoilXOnPower	torqueCoilXOnPower	0 W
torqueCoilYOnPower	torqueCoilYOnPower	0 W
torqueCoilZOnPower	torqueCoilZOnPower	0 W
Star Tracker		
averagePower	starTrackerOnPower	1.8 W
Sun Sensor		
onPower	sunSensorOnPower	1.35 W
CADET PLUS Transceiver		
rxOnPower	rxOnPower	0.7 W

rxStandbyPower	rxStandbyPower	0.3 W
maxTXdataRate	txDataRate	3200 kbps
txOnPower	txOnPower	8 W
Propulsion Subsystem		
dataProductionRate	thrusterOnDataRate	0 Bps
heatingDataRate	thrusterHeatDataRate	0 Bps
standbyDataRate	thrusterStandbyDataRate	0 Bps
SOHDataRate	thrusterSOHDataRate	0 Bps
powerStandby	thrusterStandbyPower	1 W
heatPower	thrusterHeatOn	12 W
powerFiring	thrusterPowerOn	12 W
Thruster		
thrust	thrust	12 mN
numThruster	numThruster	4
thrustUnit	thrustUnit	mN
Fuel		
mass	fuelMass	1.235 kg
specificImpulse	IspValue	40 s
IspUnit	IspUnit	s
fuelDensity	fuelDensity	1.23 $x10^{-9}$ kg/m^3
Terahertz Imaging Camera		
onPower	Payload1PowerOn	6.55 W
dataProductionRate	Payload1OnDataRate	1916.67 Bps
standbyPower	Payload1StandbyPower	1.2 W
standbyDataRate	Payload1StandbyDataRate	0 Bps
SOHDataRate	Payload1SOHDataRate	0 Bps
Software-Defined Radio for MC3		
onPower	Payload2PowerOn	5 W
dataProductionRate	Payload2OnDataRate	1100 Bps
standbyPower	Payload2standbyPower	2 W
standbyDataRate	Payload2StandbyDataRate	0 Bps
SOHDataRate	Payload2SOHDataRate	0 Bps
Satellite Path-Agnostic Communications Experiment		
onPower	Payload3PowerOn	4 W
dataProductionRate	Payload3OnDataRate	500 Bps
standbyPower	Payload3StandbyPower	1.5 W
standbyDataRate	Payload3StandbyDataRate	250 Bps
SOHDataRate	Payload3SOHDataRate	0 Bps
Payload4 Placeholder		
onPower	Payload4PowerOn	0 W
dataProductionRate	Payload4OnDataRate	0 Bps
standbyPower	Payload4StandbyPower	0 W
standbyDataRate	Payload4StandbyDataRate	0 Bps
SOHDataRate	Payload4SOHDataRate	0 Bps

Appendix B. Integration Script 1

Payload Analysis Tool (PAT) - Integration Script #1

Modified IntegrationScript.m created by Kyla Brown. The below script propagates a satellite until it reaches a desired altitude where a burn may be necessary. The script then runs the STK data through the Simulink and Stateflow diagrams found in PAT_2a.slx. The resulting data is output and parsed into a table which tells you at what time the burn should take place. The specified time is then used in IntegrationScript_2.mlx.

Parameters and Variables

Parameters to be passed from CATIA for this to work are listed below. The script can be easily tested using these variables within the code - ensure that this section is commented out when running the script from the CATIA parametric diagram.

```
%Scenario
    stkVersion = 12;  %string
    scenarioName = 'Grissom';  %string
    scenarioStartTime =  '1 Mar 2023 12:00:00.000'; %string format
    scenarioStopTime = '+1year'; %string format

%Objects
    satelliteName = 'Grissom-2'; %string
    %Ground Station
     facName = 'MC3_AFIT';  %string
     facLatitude = 39.782; %39.782;  %deg
     facLongitude = -84.0832;  %deg
     %Imager Target
    target1Name = 'Riyadh';  %string
    target1Latitude = 24.7136;  %deg
    target1Longitude = 46.6753;  %deg
     %Sensor Target
     target2Name = 'Barcelos';  %string
     target2Latitude = -0.97357;  %deg
     target2Longitude = -62.9269;  %deg
     %Extra Target
     %target3Name = 'Mbandaka';  %string
     %target3Latitude = 0.04865;  %deg
     %target3Longitude = 18.2971;  %deg

 %Initial State
 %Satellite Parameters
    dryMass = 8.765; %kg
    Cd = 2.2;
    dragArea = 0.2; %m^2
```

```
    fuelMass = 1.235; %kg
    SRPArea = 0.2; %m^2
    fuelDensity = 0.00000000123; %kg/m^3
    radiationPressureArea = 0.2; %m^2

 %Orbital Elements
     semiMajorAxis = 6828.137;  %km  (altitude is 450 km) (Earth radius
6378.137km)
     eccentricity = 0;  %km
     inclination = 60;  %deg
     argumentPeriapsis = 0;  %deg
     RAAN = 0;  %deg
     trueAnomaly = 0;  %deg

 %Scenario Parameters
    stoppingAltitude = 425; %km
    deorbitAltitude = 50; %km

%Engine Parameters
    IspValue = 40; %s
    IspUnit = 's';

    numThruster = 4;
    thrust = 25; %mN
    thrustUnit = 'mN';


%C&DH Mask Parameters
    busSOHDataRate = 9600; %bytes/sampletime
    %PayloadSOHDataRate calculated below
    miscDataProductionRate = 0; %ExtraDataRate (bytes)
    storageCapacity = 8; %GB ...converted to bytes for PAT use below
    CDHPowerOn = 2.5; %Watts
    initialMemoryStorageStatus = 1000000000; %bytes  (1GB)

%EPS Mask Values
    %Battery
    batteryInitialFill = 90; %Percent
    batteryCapacity = 6; %Ah
    battNominalVoltage = 14.4; %Volts
    chargeDischargeEfficiency = 0.995; %percent
    distroEfficiency = 0.9; %percent, power distribution efficiency
    %Solar Arrays
    PPTefficiency = 0.95; %Peak Power Tracking, percent
    solarArrayArea = 0.1702; %m^2
    solarArrayEfficiency = 0.295; %percent
```

```
    %Constraints
    DODtriggerSunSafe = 30; %percent
    DODreturnToNormalOperations = 20; %percent

%ADCS Mask Values
    magnetometerOnPower = 0.825; %Watts
    starTrackerOnPower = 1.8; %Watts
    sunSensorOnPower = 1.35; %Watts
    torqueCoilXOnPower = 0; %Watts
    torqueCoilYOnPower = 0; %Watts
    torqueCoilZOnPower = 0; %Watts

    %Reaction Wheel Values
    reactionWheelMaxPower = 1.925; %Watts
    zeroTorquePower = 0.27; %Watts
    maxMomentum = 0.015; %N-m-s
    reactionWheelMOI = 0.0010577; %kg-m^2
    initialRPM = 0; %rpm
    reactionWheelMaxRPM = 5200; %rpm

%T&C Mask Values
    rxOnPower = 0.7; %Watts
    rxStandbyPower = 0.3; %Watts
    txDataRate = 3200; %DownlinkDataRate (Kbps)
    txOnPower = 8; %Watts

%Payload
    %TIC Imager
    payload1PowerOn = 6.55;  %Watts 6.55
    payload1OnDataRate = 1916.67;  %Bytes/sampletime 1916.67
    payload1StandbyPower = 1.2;  %Watts 1.2
    payload1StandbyDataRate = 0;  %PBytes/sampletime  0
    payload1SOHDataRate = 0;  %Bytes/sampletime 0

    %SensorSDR
    payload2PowerOn = 5;  %Watts 5
    payload2OnDataRate = 1100;  %Bytes/sampletime 1100
    payload2StandbyPower = 2;  %Watts 2
    payload2StandbyDataRate = 0;  %Bytes/sampletime 0
    payload2SOHDataRate = 0;  %Bytes/sampletime 0

    %Beacon/Comms
    payload3PowerOn = 4; %Watts 4
    payload3OnDataRate = 500; %Bytes/sampletime 500
    payload3StandbyPower = 1.5; %Watts 1.5
    payload3StandbyDataRate = 250; %Bytes/sampletime 250
```

```
    payload3SOHDataRate = 0; %Byte/sampletime 0

    %Built in for future use
    payload4PowerOn = 0;  %Watts
    payload4OnDataRate = 0;  %Bytes/sampletime
    payload4StandbyPower = 0;  %Watts
    payload4StandbyDataRate = 0;  %Bytes/sampletime
    payload4SOHDataRate = 0;  %Bytes/sampletime

 %Thruster
    thrusterPowerOn = 12; %Watts
    thrusterHeatOn = 12; %Watts
    thrusterHeatDataRate = 0;
    thrusterOnDataRate = 0;
    thrusterStandbyPower = 1; %Watts
    thrusterStandbyDataRate = 0;
    thrusterSOHDataRate = 0;


    % %Other Variables
     visualizeSimulink = 1;  %toggle for visualizing PAT interface in Simulink
     visualizeResultsFigures = 1;  %toggle visualizing figures
```

Initial Calculations

```
 % Math for data rates and storage inputs from Cameo
 PayloadSOHDataRate = payload1SOHDataRate + payload2SOHDataRate +
payload3SOHDataRate + payload4SOHDataRate;
 storageCapacityConverted = storageCapacity*1000000000; %GB conversion to bytes
 batteryCapWh = batteryCapacity*battNominalVoltage; %Converstion to Wh
 BatteryDataWh = batteryCapWh*(batteryInitialFill/100); %Used in the Battery
Performance diagram

 %Math for orbital calculations
 periapsisRadiusSize = semiMajorAxis*(1-eccentricity); %km
 apoapsisRadiusSize = semiMajorAxis*(1+eccentricity); %km
```

Initiate STK Scenario

```
 CurrentDirectory = pwd;  %defines active directory on user computer where this
script resides

 global TimeStep
 global StartTime
 global EndTime
```

```
%Open STK
    app=actxserver(append('stk',string(stkVersion),'.application'));

   %Grab handle on STK root
    root = app.Personality2;

%Create New Scenario
    scenario=root.Children.New('eScenario', string(scenarioName));

%Define Start/Stop time for Scenario
    scenario.SetTimePeriod(string(scenarioStartTime),string(scenarioStopTime));
    scenario.StartTime = string(scenarioStartTime);
    scenario.StopTime = string(scenarioStopTime);

    root.ExecuteCommand('Animate * Reset');
        %The start time will be reflected in the animation tool bar.
```

Insert default satellite

Satellite properties will be changed when defining the Initial State

```
        %Insert a default satellite
    satellite=scenario.Children.New('eSatellite',string(satelliteName));
```

Insert the AFIT Ground Station

```
%Ground Station
    AFIT = scenario.Children.New('eFacility', string(facName));
    AFIT.Position.AssignGeodetic(facLatitude, facLongitude, 297);
```

Other MC3 network ground stations are found below, but will not be used in this thesis.

```
%Hard-coded MC3 network
% NPS = scenario.Children.New('eFacility', 'NPS');
%     NPS.Position.AssignGeodetic(36.597484, -121.873330, 18);
%
% SDL = scenario.Children.New('eFacility', 'SDL');
%     SDL.Position.AssignGeodetic(41.762772, -111.822452, 1401);
%
% MLB = scenario.Children.New('eFacility', 'MLB');
%     MLB.Position.AssignGeodetic(28.021662, -80.679194, 22);
%
```

```
% HSFL = scenario.Children.New('eFacility', 'HSFL');
%     HSFL.Position.AssignGeodetic(21.299185, -157.817197, 37);
%
% UNM = scenario.Children.New('eFacility', 'UNM');
%     UNM.Position.AssignGeodetic(35.084318, -106.619781, 1584);

% Future MC3 Site Locations
%USNA = scenario.Children.New('eFacility', 'USNA');
    %USNA.Position.AssignGeodetic(38.982065, -76.483940, 11);
%ASF = scenario.Children.New('eFacility', 'ASF');
    %ASF.Position.AssignGeodetic(64.859944, -147.849139, 193);
%CGA = scenario.Children.New('eFacility', 'CGA');
    %CGA.Position.AssignGeodetic(41.375949, -72.101581, 39);
%RSA = scenario.Children.New('eFacility', 'RSA');
    %RSA.Position.AssignGeodetic(34.632413, -86.595497, 170);
%PAC = scenario.Children.New('eFacility', 'PAC');
    %PAC.Position.AssignGeodetic(32.708658, -117.247249, 116);

disp('Ground Stations created...')
```

Insert ground targets

```
%----------Add Target(s)
    % Adds on target
    % Future work - develop a loop that allows user to create a defined
    % number of targets
target1 = scenario.Children.New('eTarget', string(target1Name));
    target1.Position.AssignGeodetic(string(target1Latitude),
string(target1Longitude), 0);
    disp('Target1 created...')

target2 = scenario.Children.New('eTarget', string(target2Name));
    target2.Position.AssignGeodetic(string(target2Latitude),
string(target2Longitude), 0);
    disp('Target2 created...')

% target3 = scenario.Children.New('eTarget', string(target3Name));
%     target3.Position.AssignGeodetic(string(target3Latitude),
string(target3Longitude), 0);
%     disp('Target3 created...')
```

Set-up Astrogator

```
%Change propagator to Astrogator
```

```
     satellite.SetPropagatorType('ePropagatorAstrogator')
         % Note that Astrogator satellites by default start with one Initial
State
         % and one Propagate segment

         % Create a handle to the Astrogator portion of the satellites object
model
         % for convenience
     ASTG = satellite.Propagator;
         % Create a handle to the MCS and remove all existing segments
     MCS = ASTG.MainSequence;
     MCS.RemoveAll;
```

Define the Initial State

Included below are the satellite properties of the initial state. To find the properties of your satellite in their initial state, enter the following command into the MATLAB command window: satellite.Propagator.MainSequence.Item(0).InitialState.get

To find how many modules there are and the name of each module, use the following commands:

satellite.Propagator.MainSequence.Count

satellite.Propagator.MainSequence.Item(0).Name

satellite.Propagator.MainSequence.Item(1).Name

etc. for each count...

```
 %Define Initial State
     MCS.Insert('eVASegmentTypeInitialState','Inner Orbit','-');

         % The Insert command will also return a handle to the segment it
creates
     propagate = MCS.Insert('eVASegmentTypePropagate','Propagate','-');

     %Satellite Properties
 satellite.Propagator.MainSequence.Item(0).InitialState.DryMass = dryMass; %kg
based on a 10kg 6U with 0.851kg of propellant
 satellite.Propagator.MainSequence.Item(0).InitialState.Cd = Cd;
 satellite.Propagator.MainSequence.Item(0).InitialState.DragArea = dragArea;
%m^2
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelMass = fuelMass; %kg
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelDensity =
fuelDensity; %kg/m^3
 satellite.Propagator.MainSequence.Item(0).InitialState.SRPArea = SRPArea; %m^2
```

```
 satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureArea =
radiationPressureArea; %m^2
 %satellite.Propagator.MainSequence.Item(0).InitialState.MaxFuelMass=fuelMass;
%kg
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankPressure=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankTemperature=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.Cr=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureCoeff=
;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K1=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K2=;



 %Change coordinate type to Keplerian
     % Create a handle to the Initial State Segment, set it to use Modified
     % Keplerian elements and assign new initial values
     initstate = MCS.Item('Inner Orbit');
     initstate.OrbitEpoch = scenario.StartTime;
     initstate.SetElementType('eVAElementTypeKeplerian');
     kep=initstate.Element;

  %Define COEs of Initial State
     kep.PeriapsisRadiusSize = periapsisRadiusSize;
     kep.ArgofPeriapsis= argumentPeriapsis;
     kep.Eccentricity= eccentricity;
     kep.Inclination= inclination;
     kep.RAAN= RAAN;
     kep.TrueAnomaly= trueAnomaly;
```

Insert Propagating segment

```
 % Change Propagate segment color
 Red = '0000ff';
 Green = '00ff00';
 Blue = 'ff0000';
 Cyan = 'ffff00';
 Yellow = '00ffff';
 Magenta = 'ff00ff';
 Black = '000000';
 White = 'ffffff';

 propagate.Properties.Color = uint32(hex2dec(Cyan));
```

```
 % Change the propagator type
     propagate.PropagatorName = 'Earth HPOP Default v10';

 %Define stopping condition for propagator
     propagate.StoppingConditions.Add('Duration').Properties.Trip = 7884000;
     %propagate.StoppingConditions.Add('Altitude').Properties.Trip =
stoppingAltitude; %stops propagating at altitude of 425km to prepare for the
burn
     propagate.StoppingConditions.Remove('Duration'); %removes duration stopping
condition that is automatically inserted by STK


 ASTG.RunMCS; %runs the entire sequence defined above



 %---------Set the Satellite Attitude to slew to Target Area
 % This code slews to Target 1 ONLY

 attitudePointing = satellite.Attitude.Pointing;
 attitudePointing.UseTargetPointing = 1;
 attitudePointing.Targets.RemoveAll;
 attitudePointing.Targets.Add(append('Target/',target1Name));
 attitudePointing.TargetTimes.UseAccessTimes;

 %---------Set the Satellite Attitude to optimal sun-soaking
 root.ExecuteCommand(append...
     ('SetAttitude */Satellite/',satelliteName,' Profile XPOPInertial Offset
0'));

 %---------Set the Satellite solar panels to fixed position
 root.ExecuteCommand(append...
     ('VO */Satellite/',satelliteName,' InitializeSolarPanelsToSun Enable
Off'));
```

Create Access between Satellite and Target/GS

```
 %-----STK Calculate Ground Station Access
 root.ExecuteCommand(append...
     ('Access */Satellite/',satelliteName,' */Facility/',facName,' TimePeriod
UseScenarioInterval'));

 %-----STK Calculate Target Access
 root.ExecuteCommand(append...
```

```
    ('Access */Satellite/',satelliteName,' */Target/',target1Name,' TimePeriod
UseScenarioInterval'));

 root.ExecuteCommand(append...
    ('Access */Satellite/',satelliteName,' */Target/',target2Name,' TimePeriod
UseScenarioInterval'));

 disp('Access established between Satellite with Target1, Target2 and Ground
Station...')

 %-----Was the scenario created?
    % Counts if there is an open STK scenario and sends a boolean response

 exeName = 'AgUiApplication';

 [status,result] = system('tasklist/FI "imagename eq AgUiApplication.exe"');

 num = count(result, exeName);

 if num >= 1
    Create_Scenario = true;
 else
    Create_Scenario = false;
 end

 disp('Scenario created...')
```

Report Generation

Generates reports needed to run PAT Access (target and ground), Attitude, Ephemeris, Lunar, and Sun Units for scenario set to Epoch Seconds for all reports.

```
 %-----Set the report date/time format for all exported data files:
 root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');
 root.ExecuteCommand('Units_Set * All Date EpSec ConnectReportUnitsFlag On');

 disp('Epsec Units set for PAT')

 %-----Generate LLA Report for Satellite
 root.ExecuteCommand(append...
    ('ReportCreate */Satellite/',satelliteName,' Type Export Style "LLA
Position" File "',CurrentDirectory,'\LLAPosition.csv" TimePeriod
UseScenarioInterval'));

 %-----Generate Access Report for Satellite to ground station
```

```
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportGS.csv" AccessObject
*/Facility/',facName,''));

 disp('Access to Ground Station Report Created...')

 %-----Generate Gap Report for Satellite to ground station
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsGS.csv" AccessObject */Facility/',facName,''));

 disp('Access Gaps to Ground Station Report Created...')

 %-----Generate Access Report for Satellite to Target(s)
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget1.csv" AccessObject
*/Target/',target1Name,''));

 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access to Target(s) Report Created...')

 %-----Generate Gap Report for Satellite to Target(s)
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget1.csv" AccessObject
*/Target/',target1Name,''));

 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access Gaps to Target(s) Report Created...')

 %-----Generate Ephemeris file
 root.ExecuteCommand(append...
     ('ExportDataFile */Satellite/',satelliteName,' Ephemeris
"',CurrentDirectory,'\Satellite1.e" Type STK CoordSys J2000 TimeSteps 60
CentralBody Earth'));
```

```
 disp('Ephemeris Report Created...')

 %-----Generate Attitude file
 root.ExecuteCommand(append...
     ('ExportDataFile */Satellite/',satelliteName,' Attitude
"',CurrentDirectory,'\Satellite1.a" CoordAxes J2000 TimeSteps 60 '));

 disp('Attitude Report Created...')

 %-----Generate Moon Data .txt file
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Lunar Vector
J2000" File "',CurrentDirectory,'\moon.txt" TimePeriod UseScenarioInterval'));

 disp('Moon Report Created...')

 %-----Generate Sun Data .txt file
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Sun Vector
J2000" File "',CurrentDirectory,'\sun.txt" TimePeriod UseScenarioInterval'));

 disp('Sun Report Created...')
```

MAT FILE CONVERSIONS FOR PAT

Convert Moon and Sun Files

Convert Attitude and Ephemeris Files

txt and cvs files converted to .mat files and saved in current dir.

```
 %-----Load Moon File
     fprintf(1,'Lunar Position...          ')
     moon = (readmatrix('moon.txt'))';
     save moon.mat moon
         fprintf('Done\n');
```

```
 %-----Load Sun File
     fprintf(1,'Solar Position...           ');
     sun = (readmatrix('sun.txt'))';
     save sun.mat sun
             fprintf('Done\n');
```

```
%-----Load Attitude and Ephemeris File
file_att = 'Satellite1.a';
file_eph = 'Satellite1.e';

data_att = readData(file_att);
save att.mat data_att
ephem = readData(file_eph);
save ephem.mat ephem

%------Load LLA Position File
LLA = (readmatrix('LLAPosition.csv'))';
save LLA.mat LLA

%------Load .mat files to Workspace
    load('moon.mat');
    load('sun.mat');
    load('att.mat');
    load('ephem.mat');
    load('LLA.mat');
```

READ AND PARSE ACCESS REPORTS FOR GROUND STATION

Reads Ground Station Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Ground Station access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessGS = readmatrix('AccessReportGS.csv');
accessGS(isnan(accessGS(:,1)),:) = [];
%accessGS(isnan(accessGS)) = []; %error when running this script -->
accessGS(isnan(accessGS(:,1)),:) = [];

% Using moon.mat's format, determine time-related variables:
TimeStep = moon(1,2) - moon(1,1);
StartTime = moon(1,1);
EndTime = moon(1,end-1);

% Logic check to verify that TimeStep is consistent throughout entire moon.mat
array
DiffTimeStep = max(diff(moon(1,end-1))) - min(diff(moon(1,end-1))); %If the
last TimeStep is included, the TimeStep will not be equal to 60 seconds. The
scenario time is not a perfect dividen of 60.
```

```
    if DiffTimeStep ~= 0
        fprintf('CAUTION: TimeStep may not be of equal intervals \n');
    else
        fprintf('TimeStep is...         %d seconds \n',TimeStep);
    end

% Using time-related variables, create a 1-by-X time-vector (tv)
tv = (StartTime:TimeStep:EndTime);

% Pre-allocate memory for contact vector (contact)
contact = zeros(size(tv));

% Systematically parse the Access Report and update the contact vector to
% have a "1" for durations of time when the satellite is in view of the GS
for k = 1:size(accessGS,1)
    inview = find(tv > accessGS(k,2) & tv < accessGS(k,3));
    contact(inview) = 1;
end

% Finally, combine time and contact vectors to be used by CSAT
InViewofGS = transpose([tv; contact]);
```

READ AND PARSE ACCESS REPORTS FOR TARGET

Reads Target Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Target(s) access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessTarget1 = readmatrix('AccessReportTarget1.csv');
accessTarget1(isnan(accessTarget1(:,1)),:) = [];
%accessTarget1(isnan(accessTarget1)) = []; %error when running this script -->
accessTarget1(isnan(accessTarget1(:,1)),:) = [];

accessTarget2 = readmatrix('AccessReportTarget2.csv');
accessTarget2(isnan(accessTarget2(:,1)),:) = [];
%accessTarget2(isnan(accessTarget2)) = []; %error when running this script -->
accessTarget2(isnan(accessTarget2(:,1)),:) = [];

% Pre-allocate memory for capture vector (capture)
capture1 = zeros(size(tv));
capture2 = zeros(size(tv));
```

```
% Systematically parse the Access Report and update the capture vector to
% have a "1" for durations of time when the satellite is in view of the
% target.
% However, unlike the contact vector, the payload turns on when it is
% most nadir (i.e. directly overhead) to the target. CSAT logic determines
% when to enter Standby mode and On mode.
% REFERENCE: script was first developed by Lt Heather Udell
% (see timeON.m script)
 for z = 1:size(accessTarget1,1)

     ti = accessTarget1(z,2);    % Time it enters field of view
     tf = accessTarget1(z,3);    % Time it exits field of view
     DeltaT = (tf - ti);        % Duration of pass
     nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

     % The "nadir" time needs to be rounded to the closest second
     % that is a multiple of the scenario's timestep
     nadirTimeStep = TimeStep*(round(nadir/TimeStep));

     payloadON = find(tv == nadirTimeStep);
     capture1(payloadON) = 1;

 end

  for z = 1:size(accessTarget2,1)

     ti = accessTarget2(z,2);    % Time it enters field of view
     tf = accessTarget2(z,3);    % Time it exits field of view
     DeltaT = (tf - ti);        % Duration of pass
     nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

     % The "nadir" time needs to be rounded to the closest second
     % that is a multiple of the scenario's timestep
     nadirTimeStep = TimeStep*(round(nadir/TimeStep));

     payloadON = find(tv == nadirTimeStep);
     capture2(payloadON) = 1;

  end

% Finally, combine time and capture vectors to be used by CSAT
InViewofTarget1 = transpose([tv; capture1]);
InViewofTarget2 = transpose([tv; capture2]);
```

READ AND PARSE LLA POSITION

```
LLAPosition = readmatrix('LLAPosition.csv');
Altitude = LLAPosition(2:end,4);
```

READ AND PARSE THRUSTING DATA

Creates a data set to tell CSAT when thrusting occurs for power draw.

Uses time step created above.

```
% Pre-allocate memory for thrusting vector
thrustTime = zeros(size(tv));

%There are no thrusting maneuvers in this script but "Thrusting" is still
%an input necessary for the CSAT.

% Finally, combine time and thrusting vectors to be used by CSAT
Thrusting = transpose([tv; thrustTime]);
```

CREATE SPACE FOR WORKSPACE TASK LIST GENERATED FROM CSAT

```
    % Pre-allocate memory for Workspace Task List
        WSTaskList = zeros(length(tv),15);
```

RUN THE PAT

```
    % Runs the PAT with the input values from the workspsace
    % Opens PAT for visualization (can turn on/off)

if visualizeSimulink == 1
    open_system('PAT_2a');
end

sim(['PAT_2a' ...
    '']);
```

MAKEFIGURES AND DISCRETE OUTPUTS

Outputs the makefigures.m plots

Max/Min Bus Voltage

Max/Min Solar Array Power Generation

Battery Depth of Discharge

Reaction Wheel Speed (any direction)

Max/Min Torque (any direction)

Buffer Capacity Status

```
%Plot Telemetry Data
if visualizeResultsFigures == 1
    makefigures;
end

%SA Power Generation
minSAPowerGen = min(Telem(:,20));  %Watts %This is producing the minimum Solar
Array Power Generation
maxSAPowerGen = max(Telem(:,7))  %Watts

%Total Component Power
minPowerTot = min(Telem(:,21)); %Watts %This is producing the minimum total
component power
maxPowerTot = max(Telem(:,21)); %Watts

%Depth of Discharge
maxDoD = max(Telem(:,6));  %Percent
save DoD.mat t DoD %will use for the dynamic solution

%Reaction Wheel Speed in any direction
maxRPM = max([max(Telem(:,14)), max(Telem(:,15)), max(Telem(:,16))]);  %RPM

%Max and Min Torque in any direction
maxTorque = max([max(Telem(:,17)), max(Telem(:,18)), max(Telem(:,19))]);  %N-m
minTorque = min([min(Telem(:,17)), min(Telem(:,18)), min(Telem(:,19))]);  %N-m

%Buffer (Memory Space)
maxBuffer = max(Telem(:,4));  %Percent Full
```

FAULT MODE OUTPUTS

If the fault mode was activated during scenario, the output is true

If the fault mode was not activated during scenario, the output is false

```
%RWALimit Fault Mode
RWALimit = FaultModeStatus(:,1);
if RWALimit == 0
```

```
    RWALimitStatus = false;  %no fault mode triggered
else
    RWALimitStatus = true;  %fault mode triggered
end

%Sun Safe Fault Mode
SunSafe = FaultModeStatus(:,2);
if SunSafe == 0
    SunSafeStatus = false;  %no fault mode triggered
else
    SunSafeStatus = true;  %fault mode triggered
end

%Survival Status Fault Mode
Survival = FaultModeStatus(:,3);
if Survival == 0
    SurvivalStatus = false;  %no fault mode triggered
else
    SurvivalStatus = true;  %fault mode triggered
end

%Memory Limit Fault Mode
MemoryLimit = FaultModeStatus(:,4);
if MemoryLimit == 0
    MemoryLimitStatus = false;  %no fault mode triggered
else
    MemoryLimitStatus = true;  %fault mode triggered
end
```

QUASI-STATIC SOLUTION

The quasi-static solution finds a time where all of the Sol criteria are true:

- Depth of Discharge < 20
- Altitude < 430
- Not accessing a ground station or target

```
accessGS = InViewofGS(:,2);
accessT1 = InViewofTarget1(:,2);
accessT2 = InViewofTarget2(:,2);
DynamicData = table(t,DoD,Altitude,accessGS,accessT1,accessT2);

Sol = (DynamicData.DoD<20 & DynamicData.Altitude<430 & DynamicData.accessGS==0
& DynamicData.accessT1==0 & DynamicData.accessT2==0);
```

```
DynamicData(Sol,:)
```

Attitude and Ephemeris Function to create ephem.mat and att.mat

Developed by David Evert

```
function data = readData(filename)

    fprintf('Reading data from %s...\n', filename);

    % File format based on file extension
    [~,~,extn] = fileparts(filename);
    if strcmp(extn,'.a')
        key = 'AttitudeTimeQuaternions';    % Key word that comes before data
        nskip = 0;                          % Number of lines between key word
and data
        format = '%f %f %f %f %f';          % Format of data
        nCols = 5;                          % Number of columns in data
    elseif strcmp(extn,'.e')
        key = 'EphemerisTimePosVel';        % Key word that comes before data
        nskip = 1;                          % Number of lines between key word
and data
        format = '%f %f %f %f %f %f %f';    % Format of data
        nCols = 7;                          % Number of columns in data
    end

    % Open file for reading
    fid = fopen(filename,'r+');

    % Read number of lines of data
    % Loop through file lines until it gets to NumberOf... line
    try
        flag = true;
        while flag
            currentLine = fgets(fid);
            if contains(currentLine,'NumberOf')
                flag = false;
                % Find number in current line and save to variable
                lines = str2double(currentLine(regexp(currentLine, '[\d]')));
            end
        end
    catch
        fprintf('File does not contain NumberOf... value\n');
    end
```

```
    % Find line containing keyword, then skip nskip lines after it
    try
        flag = true;
        while flag
            if contains(fgets(fid),key)
                flag = false;
                for n = 1:nskip
                    fgets(fid);
                end
            end
        end
    catch
        fprintf('File does not contain "%s" key word\n', key);
    end

    % Read data and save to .mat file
    data = fscanf(fid, format, [nCols,lines]);

    % Close file
    fclose(fid);

    fprintf('Done\n');
end
```

Data Provider Function

If additional data providers are needed for your mission, use this data provider function. It makes it very easy to pull information from STK.

```
 function [dataFull] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect)

 %Analytical Graphics, Inc.
 %Author: Mo Syed
 %Date Created: 9/12/18
 %Edit 3/11/20

 %This function takes in the data provider parameters and outputs the
 %desired data, skipping the setup that is usually needed. It automates the
 %process without the user having to get into the semantics of how data
 %providers work in object model.

 %It is also useful to have the report & graph manager open as if you were
```

```
%trying to create a custom report so you can see the possible data provider
%and proper subfolders to use as function inputs

%root is the STK root.

%dataProvString is the data provider from STK.

%times corresponds to the start, stop and step time (put in values
%accordingly). Pass in as a cell array.

%dataProvElem is the actual elements of data you are looking for. Pass in
%as a cell array. If you leave this as empty in the input (as a []), the
%function will pull ALL of the data elements in the data provider. They
%will not have the name of the data elements above them but they correspond
%to the order found in the report & graph manager in STK.

%grouping is the subfolder under the data provider type in STK. Not every
%data provider will have this so leave empty (as a []) in the inputs if
%that is the case. Pass it in as a string

%predata is the input of predata in case your dataprovider needs predata.
%Leave this empty (as a []) in the inputs if no predata is needed.

%object is the STK object the data provider is on. Pass on the handle to the
%object

%--------SPECIFIC INSTRUCTIONS FOR DATA PROVIDERS THAT NEED PREDATA--------

%You can enter in predata one of two ways into the script. The first way is
%to manually know your predata string and place that into the predata input
%listed above. The second is to allow a UI window to open up in STK that
%allows you to select the predata as you run this function. The predata
%input method allows you to keep the code streamlined, while having the UI
%window open up makes it easier for the user to select the correct predata
%with less room for error and figuring out what predata is needed. If you
%don't know if your data provider has predata, enable the option so the
%ui window will automatically popup even if your predata input is empty
%(as a [])


%To enable the noncoding UI option for predata, set the value of
%UIPredata = 1 below. If you want it off, leave it as UIPredata = 0.


%----------------Example of variable inputs for the function---------------
```

```
 %     root is your handle to Personality2
 %     dataProvString = 'Axes Choose Axes';
 %     dataProvElem = {'Time','q1','q2','q3','q4'};
 %     times = {0,2400,60};
 %     grouping = 'Body';
 %     predata = [CentralBodies/Earth];
 %     object = root.GetObjectFromPath('Satellite/Satellite1');
 %     [outputData] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect);


 %---------------------------------Code-----------------------------------

     %Grabs this to convert units properly
     scenario = root.CurrentScenario;

     %Refer to the predata readme section

     UIPredata = 0;

     %Grabs universal data provider interfaces

     dataProviderValue = object.DataProviders.Item(dataProvString);
     dataProviderType = dataProviderValue.Type;


     %Determines if time is in UTCG or EpSec

     if strcmp(class(times{1}),'double') %Epsec are double inputs

         root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');

     elseif strcmp(class(times{1}),'char') %UTCG is char inputs

         root.UnitPreferences.Item('DateFormat').SetCurrentUnit('UTCG');

     end

     %Determines if there is a subfolder on the dataprovider

     if dataProviderValue.IsGroup == 0

         objectDP = dataProviderValue;

     elseif dataProviderValue.IsGroup == 1
```

```
        try

            objectDP = dataProviderValue.Group.Item(grouping);

        catch

            error(['This data provider has a grouping input that needs to'...
                ' be assigned. Look at the report & graph manager in STK'...
                ' for the proper subfolder in the data provider that you
are'...
                ' trying to extract data for.'])

        end

    end


    %Determines if there is predata included

    objectDP.AllowUI = UIPredata;

    if ~isempty(predata)

        if UIPredata == 0

            objectDP.PreData = predata;

        end

    end


    %This is where things differ depending on report type

    switch (dataProviderType)

        case 'eDrTimeVar'

            dataProviderFinal = objectDP.Exec(times{1},times{2},times{3});

        case 'eDrIntvl'

           dataProviderFinal = objectDP.Exec(times{1},times{2});

        case 'eDrFixed'
```

```
            dataProviderFinal = objectDP.Exec;

    end

    %Determines if you want all data elements or a specified list

    dataCell = [];

     if ~isempty(dataProvElem)

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1



            for i = 1:length(dataProvElem)

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.GetDataSetByName(dataP
rovElem{i}).GetValues;
                    dataCell = [dataCell; data];

                end

                dataElem(:,i) = dataCell;
                dataCell = [];
            end

        elseif dataProviderFinal.Interval.Count == 1

            for i = 1:length(dataProvElem)

                dataElem{i} =
dataProviderFinal.DataSets.GetDataSetByName(dataProvElem{i}).GetValues;

            end

        else

                dataElem = {['No ' dataProvString ' Data Available']};
```

```
        end

     else

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.ToArray();
                    dataCell = [dataCell;  data];

                end

                dataElem = dataCell;
                dataCell = [];

        elseif dataProviderFinal.Interval.Count == 1

                dataElem = dataProviderFinal.DataSets.ToArray();

        else

                dataElem = {['No ' dataProvString ' Data Available']};

        end

     end


    for i = 1:length(dataProvElem)

        dataColumn = [dataProvElem{i} ; dataElem{i}];
        dataFull(:,i) = dataColumn;

    end



end
```

Appendix C. Integration Script 2

Payload Analysis Tool (PAT) - Integration Script #2

Modified IntegrationScript.m created by Kyla Brown. The below script propagates a satellite until it reaches a desired altitude where a burn may be necessary. The time suggested in IntegrationScript_1.mlx is then used to end the propagation and execute a first burn. The scenario ends with a transfer ellipse, preparing the satellite for a second burn.The script then runs the STK data through the Simulink and Stateflow diagrams found in PAT_2a.slx. The resulting data is output and parsed into a table which tells you how long the transfer ellipse should last before the second burn should optimally be performed. The specified time is then used in IntegrationScript_3.mlx.

Parameters and Variables

Parameters to be passed from CATIA for this to work are listed below. The script can be easily tested using these variables within the code - ensure that this section is commented out when running the script from the CATIA parametric diagram.

```
%Scenario
    stkVersion = 12;  %string
    scenarioName = 'Grissom';  %string
    scenarioStartTime =  '1 Mar 2023 12:00:00.000'; %string format
    scenarioStopTime = '+1year'; %string format

%Objects
    satelliteName = 'Grissom-2'; %string
    %Ground Station
     facName = 'MC3_AFIT';  %string
     facLatitude = 39.782; %39.782;  %deg
     facLongitude = -84.0832;  %deg
    %Imager Target
     target1Name = 'Riyadh';  %string
     target1Latitude = 24.7136;  %deg
     target1Longitude = 46.6753;  %deg
    %Sensor Target
     target2Name = 'Barcelos';  %string
     target2Latitude = -0.97357;  %deg
     target2Longitude = -62.9269;  %deg
    %Extra Target
%    target3Name = 'Mbandaka';  %string
%    target3Latitude = 0.04865;  %deg
%    target3Longitude = 18.2971;  %deg

 %Initial State
 %Satellite Parameters
    dryMass = 8.765; %kg %7.621kg without propulsion 8.765
```

```
    Cd = 2.2;
    dragArea = 0.2; %m^2
    fuelMass = 1.235; %kg 1.235
    SRPArea = 0.2; %m^2
    fuelDensity = 0.00000000123; %kg/m^3
    radiationPressureArea = 0.2; %m^2

  %Orbital Elements
     semiMajorAxis = 6828.137;  %km  (altitude is 450 km) (Earth radius
6378.137 km)
     eccentricity = 0;  %km
     inclination = 60;  %deg
     argumentPeriapsis = 0;  %deg
     RAAN = 0;  %deg
     trueAnomaly = 0;  %deg

  %Scenario Parameters
    dynamicPropStop1 = 2351280; %s %%Taken from IntegrationScript_1 dynamic
solution
    stoppingAltitude = 425; %km
    deorbitAltitude = 50; %km

 %Engine Parameters
    IspValue = 40; %s 40
    IspUnit = 's';

    numThruster = 4; %4
    thrust = 25; %mN 25
    thrustUnit = 'mN';


 %C&DH Mask Parameters
    busSOHDataRate = 9600; %bytes/sampletime
    %PayloadSOHDataRate calculated below
    miscDataProductionRate = 0; %ExtraDataRate (bytes)
    storageCapacity = 8; %GB ...converted to bytes for PAT use below
    CDHPowerOn = 2.5; %Watts
    initialMemoryStorageStatus = 1000000000; %bytes  (1GB)

 %EPS Mask Values
    %Battery
    batteryInitialFill = 90; %Percent
    batteryCapacity = 6; %Ah
    battNominalVoltage = 14.4; %Volts
    chargeDischargeEfficiency = 0.995; %percent
    distroEfficiency = 0.9; %percent, power distribution efficiency
```

```
    %Solar Arrays
    PPTefficiency = 0.95; %Peak Power Tracking, percent
    solarArrayArea = 0.1702; %m^2
    solarArrayEfficiency = 0.295; %percent
    %Constraints
    DODtriggerSunSafe = 30; %percent
    DODreturnToNormalOperations = 20; %percent

%ADCS Mask Values
    magnetometerOnPower = 0.825; %Watts
    starTrackerOnPower = 1.8; %Watts
    sunSensorOnPower = 1.35; %Watts
    torqueCoilXOnPower = 0; %Watts
    torqueCoilYOnPower = 0; %Watts
    torqueCoilZOnPower = 0; %Watts

    %Reaction Wheel Values
    reactionWheelMaxPower = 1.925; %Watts
    zeroTorquePower = 0.27; %Watts
    maxMomentum = 0.015; %N-m-s
    reactionWheelMOI = 0.0010577; %kg-m^2
    initialRPM = 0; %rpm
    reactionWheelMaxRPM = 5200; %rpm

%T&C Mask Values
    rxOnPower = 0.7; %Watts
    rxStandbyPower = 0.3; %Watts
    txDataRate = 3200; %DownlinkDataRate (Kbps)
    txOnPower = 8; %Watts

%Payload
    %TIC Imager
    payload1PowerOn = 6.55;  %Watts 6.55
    payload1OnDataRate = 1916.67;  %Bytes/sampletime 1916.67
    payload1StandbyPower = 1.2;  %Watts 1.2
    payload1StandbyDataRate = 0;  %PBytes/sampletime  0
    payload1SOHDataRate = 0;  %Bytes/sampletime 0

    %SensorSDR
    payload2PowerOn = 5;  %Watts 5
    payload2OnDataRate = 1100;  %Bytes/sampletime 1100
    payload2StandbyPower = 2;  %Watts 2
    payload2StandbyDataRate = 0;  %Bytes/sampletime 0
    payload2SOHDataRate = 0;  %Bytes/sampletime 0

    %Beacon/Comms
```

```
    payload3PowerOn = 4; %Watts 4
    payload3OnDataRate = 500; %Bytes/sampletime 500
    payload3StandbyPower = 1.5; %Watts 1.5
    payload3StandbyDataRate = 250; %Bytes/sampletime 250
    payload3SOHDataRate = 0; %Byte/sampletime 0

%     %Built in for future use
    payload4PowerOn = 0;  %Watts
    payload4OnDataRate = 0;  %Bytes/sampletime
    payload4StandbyPower = 0;  %Watts
    payload4StandbyDataRate = 0;  %Bytes/sampletime
    payload4SOHDataRate = 0;  %Bytes/sampletime

%Thruster
    thrusterPowerOn = 12; %Watts 12
    thrusterHeatOn = 12; %Watts 12
    thrusterHeatDataRate = 0;
    thrusterOnDataRate = 0;
    thrusterStandbyPower = 1; %Watts 1
    thrusterStandbyDataRate = 0;
    thrusterSOHDataRate = 0;


    % %Other Variables
     visualizeSimulink = 1;  %toggle for visualizing PAT interface in Simulink
     visualizeResultsFigures = 1;  %toggle visualizing figures
```

Initial Calculations

```
% Math for data rates and storage inputs from Cameo
PayloadSOHDataRate = payload1SOHDataRate + payload2SOHDataRate +
payload3SOHDataRate + payload4SOHDataRate;
storageCapacityConverted = storageCapacity*1000000000; %GB conversion to bytes
batteryCapWh = batteryCapacity*battNominalVoltage; %Converstion to Wh
BatteryDataWh = batteryCapWh*(batteryInitialFill/100); %Used in the Battery
Performance diagram

%Math for orbital calculations
periapsisRadiusSize = semiMajorAxis*(1-eccentricity); %km
apoapsisRadiusSize = semiMajorAxis*(1+eccentricity); %km
```

Initiate STK Scenario

```
 CurrentDirectory = pwd;  %defines active directory on user computer where this
script resides

 global TimeStep
 global StartTime
 global EndTime

 %Open STK
     app=actxserver(append('stk',string(stkVersion),'.application'));

    %Grab handle on STK root
     root = app.Personality2;

 %Create New Scenario
     scenario=root.Children.New('eScenario', string(scenarioName));

 %Define Start/Stop time for Scenario
     scenario.SetTimePeriod(string(scenarioStartTime),string(scenarioStopTime));
     root.ExecuteCommand('Animate * Reset');
         %The start time will be reflected in the animation tool bar.
```

Insert default satellite

Satellite properties will be changed when defining the Initial State

```
 %Insert a default satellite
     satellite=scenario.Children.New('eSatellite',string(satelliteName));
```

Insert the AFIT Ground Station

```
 %Ground Station
     AFIT = scenario.Children.New('eFacility', string(facName));
     AFIT.Position.AssignGeodetic(facLatitude, facLongitude, 297);
```

Other MC3 network ground stations are found below, but will not be used in this thesis.

```
 %Hard-coded MC3 network
 % NPS = scenario.Children.New('eFacility', 'NPS');
 %     NPS.Position.AssignGeodetic(36.597484, -121.873330, 18);
 %
 % SDL = scenario.Children.New('eFacility', 'SDL');
 %     SDL.Position.AssignGeodetic(41.762772, -111.822452, 1401);
```

```
%
% MLB = scenario.Children.New('eFacility', 'MLB');
%     MLB.Position.AssignGeodetic(28.021662, -80.679194, 22);
%
% HSFL = scenario.Children.New('eFacility', 'HSFL');
%     HSFL.Position.AssignGeodetic(21.299185, -157.817197, 37);
%
% UNM = scenario.Children.New('eFacility', 'UNM');
%     UNM.Position.AssignGeodetic(35.084318, -106.619781, 1584);

% Future MC3 Site Locations
%USNA = scenario.Children.New('eFacility', 'USNA');
    %USNA.Position.AssignGeodetic(38.982065, -76.483940, 11);
%ASF = scenario.Children.New('eFacility', 'ASF');
    %ASF.Position.AssignGeodetic(64.859944, -147.849139, 193);
%CGA = scenario.Children.New('eFacility', 'CGA');
    %CGA.Position.AssignGeodetic(41.375949, -72.101581, 39);
%RSA = scenario.Children.New('eFacility', 'RSA');
    %RSA.Position.AssignGeodetic(34.632413, -86.595497, 170);
%PAC = scenario.Children.New('eFacility', 'PAC');
    %PAC.Position.AssignGeodetic(32.708658, -117.247249, 116);

disp('Ground Stations created...')
```

Insert ground targets

```
%----------Add Target(s)
    % Adds on target
    % Future work - develop a loop that allows user to create a defined
    % number of targets

target1 = scenario.Children.New('eTarget', string(target1Name));
    target1.Position.AssignGeodetic(string(target1Latitude),
string(target1Longitude), 0);
    disp('Target1 created...')

target2 = scenario.Children.New('eTarget', string(target2Name));
    target2.Position.AssignGeodetic(string(target2Latitude),
string(target2Longitude), 0);
    disp('Target2 created...')

% target3 = scenario.Children.New('eTarget', string(target3Name));
%     target3.Position.AssignGeodetic(string(target3Latitude),
string(target3Longitude), 0);
```

```
 %     disp('Target3 created...')
```

Set-up Astrogator

```
 %-----Change propagator to Astrogator
     satellite.SetPropagatorType('ePropagatorAstrogator')
         % Note that Astrogator satellites by default start with one Initial
State
         % and one Propagate segment

         % Create a handle to the Astrogator portion of the satellites object
model
         % for convenience
     ASTG = satellite.Propagator;
         % Create a handle to the MCS and remove all existing segments
     MCS = ASTG.MainSequence;
     MCS.RemoveAll;
```

Define the Initial State

Included below are the satellite properties of the initial state. To find the properties of your satellite in their initial state, enter the following command into the MATLAB command window: satellite.Propagator.MainSequence.Item(0).InitialState.get

To find how many modules there are and the name of each module, use the following commands:

satellite.Propagator.MainSequence.Count

satellite.Propagator.MainSequence.Item(0).Name

satellite.Propagator.MainSequence.Item(1).Name

etc. for each count...

```
 %-----Define Initial State
     MCS.Insert('eVASegmentTypeInitialState','Inner Orbit','-');

         % The Insert command will also return a handle to the segment it
creates
     propagate = MCS.Insert('eVASegmentTypePropagate','Propagate','-');

 %Satellite Properties
 satellite.Propagator.MainSequence.Item(0).InitialState.DryMass = dryMass; %kg
based on a 10kg 6U with 0.851kg of propellant
 satellite.Propagator.MainSequence.Item(0).InitialState.Cd = Cd;
 satellite.Propagator.MainSequence.Item(0).InitialState.DragArea = dragArea;
%m^2
```

```
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelMass = fuelMass; %kg
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelDensity =
fuelDensity; %kg/m^3
 satellite.Propagator.MainSequence.Item(0).InitialState.SRPArea = SRPArea; %m^2
 satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureArea =
radiationPressureArea; %m^2
 %satellite.Propagator.MainSequence.Item(0).InitialState.MaxFuelMass=fuelMass;
%kg
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankPressure=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankTemperature=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.Cr=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureCoeff=
;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K1=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K2=;

 %-----Change coordinate type to Keplerian
     % Create a handle to the Initial State Segment, set it to use Modified
     % Keplerian elements and assign new initial values
     initstate = MCS.Item('Inner Orbit');
     initstate.OrbitEpoch = scenario.StartTime;
     initstate.SetElementType('eVAElementTypeKeplerian');
     kep=initstate.Element;

 %-----Define COEs of Initial State
     kep.PeriapsisRadiusSize = periapsisRadiusSize;
     kep.ArgofPeriapsis= argumentPeriapsis;
     kep.Eccentricity= eccentricity;
     kep.Inclination= inclination;
     kep.RAAN= RAAN;
     kep.TrueAnomaly= trueAnomaly;
```

Insert Propagating segment

```
 %-----Change Propagate segment color
 Red = '0000ff';
 Green = '00ff00';
 Blue = 'ff0000';
 Cyan = 'ffff00';
 Yellow = '00ffff';
 Magenta = 'ff00ff';
 Black = '000000';
 White = 'ffffff';

 propagate.Properties.Color = uint32(hex2dec(Cyan));
```

```
 %-----Change the propagator type
     propagate.PropagatorName = 'Earth HPOP Default v10';

 %-----Define stopping condition for propagator
      propagate.StoppingConditions.Add('Duration').Properties.Trip =
dynamicPropStop1; %stops propagating at time specified by the dynamic solution
in part 1
      propagate.StoppingConditions.Remove('Duration'); %removes duration
stopping condition
```

Insert Target Sequence

Create custom engine

Insert target sequence

Define first burn

```
 %-----Create Custom Engine
 %Isp Value conversions
 Ispstr = num2str(IspValue);
 Isp = append(Ispstr, ' ', IspUnit);

 %Thrust Value conversions
  totalThrust = numThruster * thrust;
  totalThruststr = num2str(totalThrust);
  finalThrust = append(totalThruststr, ' ', thrustUnit);

 %Create a new engine model
  root.ExecuteCommand(append('ComponentBrowser */ Duplicate "Engine Models"
"Constant Thrust and Isp" "Cold Gas Thruster"'));
  root.ExecuteCommand(append('ComponentBrowser */ SetValue "Engine Models" "Cold
Gas Thruster" Isp ', Isp ));
  root.ExecuteCommand(append('ComponentBrowser */ SetValue "Engine Models" "Cold
Gas Thruster" Thrust ', finalThrust));

 %-----Insert target sequence (ex: raising the orbit)
      % Insert a Target Sequence with a nested Maneuver segment
         ts = MCS.Insert('eVASegmentTypeTargetSequence','Start Transfer','-');
         dv1 = ts.Segments.Insert('eVASegmentTypeManeuver','DV1','-');
%inserting the first maneuver/first burn
         dv1.Properties.Color = uint32(hex2dec(Red)); %change color in STK

 %-----Define the first burn within the Target Sequence
```

```
     dv1.SetManeuverType('eVAManeuverTypeImpulsive'); %sets maneuver to an
impulsive burn
         % Create a handle to the impulsive properties of the maneuver
         impulsive = dv1.Maneuver;
         impulsive.SetAttitudeControlType('eVAAttitudeControlThrustVector');
%attitude control employed in thrust vector
         impulsive.UpdateMass = true; %updates satellite mass as fuel is burned
         impulsive.SetPropulsionMethod('eVAPropulsionMethodEngineModel', 'Cold
Gas Thruster');

         % Create a handle to the Attitude Control - Thrust Vector properties of
the
         % maneuver and set the appropriate axes
         thrustVector = impulsive.AttitudeControl;
         thrustVector.ThrustAxesName = 'Satellite VNC(Earth)';

         dv1.EnableControlParameter('eVAControlManeuverImpulsiveCartesianX');
         dv1.Results.Add('Keplerian Elems/Radius of Apoapsis');

         %Handle to differential corrector profile
         dc=ts.Profiles.Item('Differential Corrector');

         %Set up control parameter

xControlParam=dc.ControlParameters.GetControlByPaths('DV1','ImpulsiveMnvr.Cartes
ian.X');
         xControlParam.Enable=true;
         xControlParam.MaxStep=0.3;

         %Set up result for control parameter
         roaResult = dc.Results.GetResultByPaths('DV1', 'Radius Of Apoapsis');
         roaResult.Enable=true;
         roaResult.DesiredValue = apoapsisRadiusSize;
         roaResult.Tolerance=0.1;

         %Set final differential corrector and targeter properties and run modes
         dc.MaxIterations=50;
         dc.EnableDisplayStatus=true;
         dc.Mode='eVAProfileModeIterate';
         ts.Action='eVATargetSeqActionRunActiveProfiles';
```

Insert transfer ellipse

Transfer ellipse will propagate after the first burn until the satellite reaches its desired position for the second burn.

```
 %-----Propagate the Transfer Orbit to Apogee to prep for second burn
     transferEllipse=MCS.Insert('eVASegmentTypePropagate' , 'Transfer
Ellipse','-');
     transferEllipse.PropagatorName='Earth Point Mass'; %uses the Earth Point
Mass propagator, due to small amount of time spent propagating

 %-----Add an Stopping Condition and remove the Duration Stopping Condition
    transferEllipse.StoppingConditions.Add('Duration').Properties.Trip = 20000;
%s %20,000s = ~3.5 orbits
   %  transferEllipse.StoppingConditions.Add('Apoapsis'); %stop propagating
   %  at apoapsis --> Not needed for this code, but at times helpful when
   %  struggling to find a good time for the second burn.
     transferEllipse.StoppingConditions.Remove('Duration'); %remove duration
stopping condition automatically inserted by STK


 transferEllipse.Results.Add('Keplerian Elems/Radius of Apoapsis'); %enables STK
to produce the radius of apoapsis value during the transfer ellipse

 ASTG.RunMCS; %runs the entire sequence defined above

 %-----Calculates the radius of apoapsis, which should be close tot 450km. This
 %is where the second burn should take place.
 RoA = transferEllipse.GetResultValue('Radius of Apoapsis');
 apoAlt = RoA - 6378;

 disp(['Transfer Ellipse Apoapsis:'
num2str(transferEllipse.GetResultValue('Radius of Apoapsis'))]);

 %---------Set the Satellite Attitude to slew to Target Area
     % This code slews to Target 1 ONLY

 attitudePointing = satellite.Attitude.Pointing;
 attitudePointing.UseTargetPointing = 1;
 attitudePointing.Targets.RemoveAll;
 attitudePointing.Targets.Add(append('Target/',target1Name));
 attitudePointing.TargetTimes.UseAccessTimes;

 %---------Set the Satellite Attitude to optimal sun-soaking
 root.ExecuteCommand(append...
     ('SetAttitude */Satellite/',satelliteName,' Profile XPOPInertial Offset
0'));

 %---------Set the Satellite solar panels to fixed position
 root.ExecuteCommand(append...
```

```
    ('VO */Satellite/',satelliteName,' InitializeSolarPanelsToSun Enable
Off'));
```

Create Access between Satellite and Target/GS

```
 %-----STK Calculate Ground Station Access
 root.ExecuteCommand(append...
    ('Access */Satellite/',satelliteName,' */Facility/',facName,' TimePeriod
UseScenarioInterval'));

 %-----STK Calculate Target Access
 root.ExecuteCommand(append...
    ('Access */Satellite/',satelliteName,' */Target/',target1Name,' TimePeriod
UseScenarioInterval'));

 root.ExecuteCommand(append...
    ('Access */Satellite/',satelliteName,' */Target/',target2Name,' TimePeriod
UseScenarioInterval'));

 disp('Access established between Satellite with Target1, Target2 and Ground
Station...')

 %-----Was the scenario created?
    % Counts if there is an open STK scenario and sends a boolean response

 exeName = 'AgUiApplication';

 [status,result] = system('tasklist/FI "imagename eq AgUiApplication.exe"');

 num = count(result, exeName);

 if num >= 1
    Create_Scenario = true;
 else
    Create_Scenario = false;
 end

 disp('Scenario created...')
```

Report Generation

Generates reports needed to run PAT Access (target and ground), Attitude, Ephemeris, Lunar, and Sun Units for scenario set to Epoch Seconds for all reports.

```
 %-----Set the report date/time format for all exported data files:
 root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');
 root.ExecuteCommand('Units_Set * All Date EpSec ConnectReportUnitsFlag On');

 disp('Epsec Units set for PAT')

 %-----Generate LLA Report for Satellite to later determine altitude
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "LLA
Position" File "',CurrentDirectory,'\LLAPosition.csv" TimePeriod
UseScenarioInterval'));

 %-----Generate Access Report for Satellite to ground station
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportGS.csv" AccessObject
*/Facility/',facName,''));

 disp('Access to Ground Station Report Created...')

 %-----Generate Gap Report for Satellite to ground station
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsGS.csv" AccessObject */Facility/',facName,''));

 disp('Access Gaps to Ground Station Report Created...')

 %-----Generate Access Report for Satellite to Target(s)
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget1.csv" AccessObject
*/Target/',target1Name,''));

 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access to Target(s) Report Created...')

 %-----Generate Gap Report for Satellite to Target(s)
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget1.csv" AccessObject
*/Target/',target1Name,''));
```

```
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access Gaps to Target(s) Report Created...')

 %-----Generate Ephemeris file
 root.ExecuteCommand(append...
     ('ExportDataFile */Satellite/',satelliteName,' Ephemeris
"',CurrentDirectory,'\Satellite1.e" Type STK CoordSys J2000 TimeSteps 60
CentralBody Earth'));

 disp('Ephemeris Report Created...')

 %-----Generate Attitude file
 root.ExecuteCommand(append...
     ('ExportDataFile */Satellite/',satelliteName,' Attitude
"',CurrentDirectory,'\Satellite1.a" CoordAxes J2000 TimeSteps 60 '));

 disp('Attitude Report Created...')

 %-----Generate Moon Data .txt file
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Lunar Vector
J2000" File "',CurrentDirectory,'\moon.txt" TimePeriod UseScenarioInterval'));

 disp('Moon Report Created...')

 %-----Generate Sun Data .txt file
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Sun Vector
J2000" File "',CurrentDirectory,'\sun.txt" TimePeriod UseScenarioInterval'));

 disp('Sun Report Created...')

 %-----Generate Maneuver Data .txt file
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Maneuver
Summary" File "',CurrentDirectory,'\maneuver.txt"'));
 %Maneuver Summary only includes DeltaV and Fuel Used, it does not include
 %the semimajor axis information, that is why you need the LLAPosition data.

 disp('Maneuver Report Created...')
```

MAT FILE CONVERSIONS FOR PAT

Convert Moon and Sun Files

Convert Attitude and Ephemeris Files

txt and cvs files converted to .mat files and saved in current dir.

```
%-----Load Moon File
    fprintf(1,'Lunar Position...          ')
    moon = (readmatrix('moon.txt'))';
    save moon.mat moon
        fprintf('Done\n');

%-----Load Sun File
    fprintf(1,'Solar Position...           ');
    sun = (readmatrix('sun.txt'))';
    save sun.mat sun
            fprintf('Done\n');

%-----Load Attitude and Ephemeris File
file_att = 'Satellite1.a';
file_eph = 'Satellite1.e';

data_att = readData(file_att);
save att.mat data_att
ephem = readData(file_eph);
save ephem.mat ephem

%------Load LLA Position File
LLA = (readmatrix('LLAPosition.csv'))';
save LLA.mat LLA

%-----Load Maneuver File
    fprintf(1,'Maneuver...                  ');
    maneuver = (readmatrix('maneuver.txt'))';
    maneuver(isnan(maneuver(:,1)),:) = [];
    save maneuver.mat maneuver
            fprintf('Done\n');

%------Load .mat files to Workspace
    load('moon.mat');
    load('sun.mat');
    load('maneuver.mat');
    load('att.mat');
    load('ephem.mat');
```

```
    load('LLA.mat');
```

READ AND PARSE ACCESS REPORTS FOR GROUND STATION

Reads Ground Station Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Ground Station access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessGS = readmatrix('AccessReportGS.csv');
accessGS(isnan(accessGS(:,1)),:) = [];
%accessGS(isnan(accessGS)) = []; %use if there is an error when running this
script --> accessGS(isnan(accessGS(:,1)),:) = [];

% Using moon.mat's format, determine time-related variables:
TimeStep = moon(1,2) - moon(1,1);
StartTime = moon(1,1);
EndTime = moon(1,end-1);

% Logic check to verify that TimeStep is consistent throughout entire moon.mat
array
DiffTimeStep = max(diff(moon(1,end-1))) - min(diff(moon(1,end-1))); %If the
last TimeStep is included, the TimeStep will not be equal to 60 seconds. The
scenario time is not a perfect dividen of 60.
    if DiffTimeStep ~= 0
        fprintf('CAUTION: TimeStep may not be of equal intervals \n');
    else
        fprintf('TimeStep is...         %d seconds \n',TimeStep);
    end

% Using time-related variables, create a 1-by-X time-vector (tv)
tv = (StartTime:TimeStep:EndTime);

% Pre-allocate memory for contact vector (contact)
contact = zeros(size(tv));

% Systematically parse the Access Report and update the contact vector to
% have a "1" for durations of time when the satellite is in view of the GS
for k = 1:size(accessGS,1)
    inview = find(tv > accessGS(k,2) & tv < accessGS(k,3));
    contact(inview) = 1;
end
```

```
% Finally, combine time and contact vectors to be used by CSAT
InViewofGS = transpose([tv; contact]);
```

READ AND PARSE ACCESS REPORTS FOR TARGET

Reads Target Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Target(s) access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessTarget1 = readmatrix('AccessReportTarget1.csv');
accessTarget1(isnan(accessTarget1(:,1)),:) = [];

accessTarget2 = readmatrix('AccessReportTarget2.csv');
accessTarget2(isnan(accessTarget2(:,1)),:) = [];

% Pre-allocate memory for capture vector (capture)
capture1 = zeros(size(tv));
capture2 = zeros(size(tv));
```

Systematically parse the Access Report and update the capture vector to have a "1" for durations of time when the satellite is in view of the target.

However, unlike the contact vector, the payload turns on when it is most nadir (i.e. directly overhead) to the target. CSAT logic determines when to enter Standby mode and On mode.

REFERENCE: script was first developed by Lt Heather Udell (see timeON.m script)

```
 for z = 1:size(accessTarget1,1)

     ti = accessTarget1(z,2);    % Time it enters field of view
     tf = accessTarget1(z,3);    % Time it exits field of view
     DeltaT = (tf - ti);        % Duration of pass
     nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

     % The "nadir" time needs to be rounded to the closest second
     % that is a multiple of the scenario's timestep
     nadirTimeStep = TimeStep*(round(nadir/TimeStep));

     payloadON = find(tv == nadirTimeStep);
     capture1(payloadON) = 1;
```

```
 end

  for z = 1:size(accessTarget2,1)

     ti = accessTarget2(z,2);    % Time it enters field of view
     tf = accessTarget2(z,3);    % Time it exits field of view
     DeltaT = (tf - ti);        % Duration of pass
     nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

     % The "nadir" time needs to be rounded to the closest second
     % that is a multiple of the scenario's timestep
     nadirTimeStep = TimeStep*(round(nadir/TimeStep));

     payloadON = find(tv == nadirTimeStep);
     capture2(payloadON) = 1;

  end

% Finally, combine time and capture vectors to be used by CSAT
InViewofTarget1 = transpose([tv; capture1]);
InViewofTarget2 = transpose([tv; capture2]);
```

READ AND PARSE LLA POSITION

```
LLAPosition = readmatrix('LLAPosition.csv');
Altitude = LLAPosition(2:end,4);
```

READ AND PARSE THRUSTING DATA

Creates a data set to tell CSAT when thrusting occurs for power draw.

Uses time step created above.

```
% Pre-allocate memory for thrusting vector
thrustTime = zeros(size(tv));
```

Systematically update the thrusting vector to have a "2" for durations of time when the satellite propulsion system in heating and a "1" for durations of time when the satellite propulsion system is thrusting. The heating duration is set to 1800 seconds generically, but should be modified to be specific to the heating necessary for the specific propulsion system being used.

```
heatingDuration = 1800; %s
heatingStart = dynamicPropStop1 - heatingDuration;
```

```
 heatingEnd = dynamicPropStop1 - 60;


 for k = 1:size(tv,1)
     heating = find(tv < dynamicPropStop1 & tv > dynamicPropStop1 -
heatingDuration);
     thrustTime(heating) = 2;

     thrustTime1 = maneuver(5,1);
     %       %This is the most consistent way to find the thrusting time,
     %       but at times this produces an error. If you have an error, run
     %       the script to produce the maneuver file. Run the script a
     %       second time, with an inserted thrustTime1 based on the Est./Act.
     %       Finite Burn Duration (sec) information.
     thrusting = find(tv >= dynamicPropStop1 & tv < dynamicPropStop1 +
thrustTime1); % thrusting occurs after dynamicPropStop1 and continues for the
duration of the burn
     thrustTime(thrusting) = 1;
 end


 % Finally, combine time and thrusting vectors to be used by CSAT
 Thrusting = transpose([tv; thrustTime]);
```

CREATE SPACE FOR WORKSPACE TASK LIST GENERATED FROM CSAT

```
 % Pre-allocate memory for Workspace Task List
         WSTaskList = zeros(length(tv),15);
```

RUN THE PAT

```
     % Runs the PAT with the input values from the workspsace
     % Opens PAT for visualization (can turn on/off)

 if visualizeSimulink == 1
     open_system('PAT_2a');
 end

 sim(['PAT_2a' ...
     '']);
```

MAKEFIGURES AND DISCRETE OUTPUTS

Outputs the makefigures.m plots

Max/Min Bus Voltage

Max/Min Solar Array Power Generation

Battery Depth of Discharge

Reaction Wheel Speed (any direction)

Max/Min Torque (any direction)

Buffer Capacity Status

```
%Plot Telemetry Data
if visualizeResultsFigures == 1
    makefigures;
end

%SA Power Generation
minSAPowerGen = min(Telem(:,20));  %Watts %This is producing the minimum Solar
Array Power Generation
maxSAPowerGen = max(Telem(:,7));  %Watts

minPowerTot = min(Telem(:,21)); %Watts %This is producing the minimum total
component power
maxPowerTot = max(Telem(:,21));

%Depth of Discharge
maxDoD = max(Telem(:,6));  %percent
save DoD.mat t DoD

%Reaction Wheel Speed in any direction
maxRPM = max([max(Telem(:,14)), max(Telem(:,15)), max(Telem(:,16))]);  %RPM

%Max and Min Torque in any direction
maxTorque = max([max(Telem(:,17)), max(Telem(:,18)), max(Telem(:,19))]);  %N-m
minTorque = min([min(Telem(:,17)), min(Telem(:,18)), min(Telem(:,19))]);  %N-m

%Buffer (Memory Space)
maxBuffer = max(Telem(:,4));  %Percent Full
```

FAULT MODE OUTPUTS

If the fault mode was activated during scenario, the output is true

If the fault mode was not activated during scenario, the output is false

```
%RWALimit Fault Mode
```

```
RWALimit = FaultModeStatus(:,1);
if RWALimit == 0
    RWALimitStatus = false;  %no fault mode triggered
else
    RWALimitStatus = true;  %fault mode triggered
end

%Sun Safe Fault Mode
SunSafe = FaultModeStatus(:,2);
if SunSafe == 0
    SunSafeStatus = false;  %no fault mode triggered
else
    SunSafeStatus = true;  %fault mode triggered
end

%Survival Status Fault Mode
Survival = FaultModeStatus(:,3);
if Survival == 0
    SurvivalStatus = false;  %no fault mode triggered
else
    SurvivalStatus = true;  %fault mode triggered
end

%Memory Limit Fault Mode
MemoryLimit = FaultModeStatus(:,4);
if MemoryLimit == 0
    MemoryLimitStatus = false;  %no fault mode triggered
else
    MemoryLimitStatus = true;  %fault mode triggered
end
```

QUASI-STATIC SOLUTION

The quasi-static solution finds a time for the second burn where all of the Sol criteria are true:

- Depth of Discharge < 20
- Altitude is +/- 0.1km of the apoapsis
- Not accessing a ground station or target

```
accessGS = InViewofGS(:,2);
accessT1 = InViewofTarget1(:,2);
accessT2 = InViewofTarget2(:,2);
DynamicData = table(t,DoD,Altitude,accessGS,accessT1,accessT2);
minapoAlt = apoAlt - 0.1;
maxapoAlt = apoAlt + 0.1;
```

```
 Sol = (DynamicData.t>dynamicPropStop1 & DynamicData.DoD<20 &
DynamicData.Altitude>minapoAlt & DynamicData.Altitude<maxapoAlt &
DynamicData.accessGS==0 & DynamicData.accessT1==0 & DynamicData.accessT2==0);
 DynamicData(Sol,:)

 %Use the suggested time for the dynamicPropStop2 needed in IntegrationScript_3.
 %Test out multiple different times to find which time produces the lowest
deltaV
 %in order to save fuel and extend mission life.
```

Attitude and Ephemeris Function to create ephem.mat and att.mat

Developed by David Evert

```
 function data = readData(filename)

     fprintf('Reading data from %s...\n', filename);

     % File format based on file extension
     [~,~,extn] = fileparts(filename);
     if strcmp(extn,'.a')
         key = 'AttitudeTimeQuaternions';    % Key word that comes before data
         nskip = 0;                          % Number of lines between key word
and data
         format = '%f %f %f %f %f';          % Format of data
         nCols = 5;                          % Number of columns in data
     elseif strcmp(extn,'.e')
         key = 'EphemerisTimePosVel';        % Key word that comes before data
         nskip = 1;                          % Number of lines between key word
and data
         format = '%f %f %f %f %f %f %f';    % Format of data
         nCols = 7;                          % Number of columns in data
     end

     % Open file for reading
     fid = fopen(filename,'r+');

     % Read number of lines of data
     % Loop through file lines until it gets to NumberOf... line
     try
         flag = true;
         while flag
             currentLine = fgets(fid);
             if contains(currentLine,'NumberOf')
                 flag = false;
```

```
                % Find number in current line and save to variable
                lines = str2double(currentLine(regexp(currentLine, '[\d]')));
            end
        end
    catch
        fprintf('File does not contain NumberOf... value\n');
    end

    % Find line containing keyword, then skip nskip lines after it
    try
        flag = true;
        while flag
            if contains(fgets(fid),key)
                flag = false;
                for n = 1:nskip
                    fgets(fid);
                end
            end
        end
    catch
        fprintf('File does not contain "%s" key word\n', key);
    end

    % Read data and save to .mat file
    data = fscanf(fid, format, [nCols,lines]);

    % Close file
    fclose(fid);

    fprintf('Done\n');
end
```

Data Provider Function

If additional data providers are needed for your mission, use this data provider function. It makes it very easy to pull information from STK.

```
 function [dataFull] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect)

 %Analytical Graphics, Inc.
 %Author: Mo Syed
 %Date Created: 9/12/18
 %Edit 3/11/20
```

```
%This function takes in the data provider parameters and outputs the
%desired data, skipping the setup that is usually needed. It automates the
%process without the user having to get into the semantics of how data
%providers work in object model.

%It is also useful to have the report & graph manager open as if you were
%trying to create a custom report so you can see the possible data provider
%and proper subfolders to use as function inputs

%root is the STK root.

%dataProvString is the data provider from STK.

%times corresponds to the start, stop and step time (put in values
%accordingly). Pass in as a cell array.

%dataProvElem is the actual elements of data you are looking for. Pass in
%as a cell array. If you leave this as empty in the input (as a []), the
%function will pull ALL of the data elements in the data provider. They
%will not have the name of the data elements above them but they correspond
%to the order found in the report & graph manager in STK.

%grouping is the subfolder under the data provider type in STK. Not every
%data provider will have this so leave empty (as a []) in the inputs if
%that is the case. Pass it in as a string

%predata is the input of predata in case your dataprovider needs predata.
%Leave this empty (as a []) in the inputs if no predata is needed.

%object is the STK object the data provider is on. Pass on the handle to the
%object

%--------SPECIFIC INSTRUCTIONS FOR DATA PROVIDERS THAT NEED PREDATA--------

%You can enter in predata one of two ways into the script. The first way is
%to manually know your predata string and place that into the predata input
%listed above. The second is to allow a UI window to open up in STK that
%allows you to select the predata as you run this function. The predata
%input method allows you to keep the code streamlined, while having the UI
%window open up makes it easier for the user to select the correct predata
%with less room for error and figuring out what predata is needed. If you
%don't know if your data provider has predata, enable the option so the
%ui window will automatically popup even if your predata input is empty
%(as a [])
```

```
%To enable the noncoding UI option for predata, set the value of
%UIPredata = 1 below. If you want it off, leave it as UIPredata = 0.


%----------------Example of variable inputs for the function---------------

%     root is your handle to Personality2
%     dataProvString = 'Axes Choose Axes';
%     dataProvElem = {'Time','q1','q2','q3','q4'};
%     times = {0,2400,60};
%     grouping = 'Body';
%     predata = [CentralBodies/Earth];
%     object = root.GetObjectFromPath('Satellite/Satellite1');
%     [outputData] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect);


%---------------------------------Code-----------------------------------

    %Grabs this to convert units properly
    scenario = root.CurrentScenario;

    %Refer to the predata readme section

    UIPredata = 0;

    %Grabs universal data provider interfaces

    dataProviderValue = object.DataProviders.Item(dataProvString);
    dataProviderType = dataProviderValue.Type;


    %Determines if time is in UTCG or EpSec

    if strcmp(class(times{1}),'double') %Epsec are double inputs

        root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');

    elseif strcmp(class(times{1}),'char') %UTCG is char inputs

        root.UnitPreferences.Item('DateFormat').SetCurrentUnit('UTCG');

    end
```

```
    %Determines if there is a subfolder on the dataprovider

    if dataProviderValue.IsGroup == 0

        objectDP = dataProviderValue;

    elseif dataProviderValue.IsGroup == 1

        try

            objectDP = dataProviderValue.Group.Item(grouping);

        catch

            error(['This data provider has a grouping input that needs to'...
                ' be assigned. Look at the report & graph manager in STK'...
                ' for the proper subfolder in the data provider that you
are'...
                ' trying to extract data for.'])

        end

    end


    %Determines if there is predata included

    objectDP.AllowUI = UIPredata;

    if ~isempty(predata)

        if UIPredata == 0

            objectDP.PreData = predata;

        end

    end


    %This is where things differ depending on report type

    switch (dataProviderType)

        case 'eDrTimeVar'
```

```
            dataProviderFinal = objectDP.Exec(times{1},times{2},times{3});

        case 'eDrIntvl'

           dataProviderFinal = objectDP.Exec(times{1},times{2});

        case 'eDrFixed'

           dataProviderFinal = objectDP.Exec;

    end

    %Determines if you want all data elements or a specified list

    dataCell = [];

     if ~isempty(dataProvElem)

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1



            for i = 1:length(dataProvElem)

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.GetDataSetByName(dataP
rovElem{i}).GetValues;
                    dataCell = [dataCell; data];

                end

                dataElem(:,i) = dataCell;
                dataCell = [];
            end

        elseif dataProviderFinal.Interval.Count == 1

            for i = 1:length(dataProvElem)
```

```
                dataElem{i} =
dataProviderFinal.DataSets.GetDataSetByName(dataProvElem{i}).GetValues;

            end

        else

                dataElem = {['No ' dataProvString ' Data Available']};

        end

      else

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.ToArray();
                    dataCell = [dataCell;  data];

                end

                dataElem = dataCell;
                dataCell = [];

        elseif dataProviderFinal.Interval.Count == 1

                dataElem = dataProviderFinal.DataSets.ToArray();

        else

                dataElem = {['No ' dataProvString ' Data Available']};

        end

      end


     for i = 1:length(dataProvElem)

        dataColumn = [dataProvElem{i} ; dataElem{i}];
```

```
        dataFull(:,i) = dataColumn;

    end


end
```

Appendix D. Integration Script 3

Payload Analysis Tool (PAT) - Integration Script #3

Modified IntegrationScript.m created by Kyla Brown. The below script propagates a satellite until it reaches a desired altitude where a burn may be necessary. The time suggested in IntegrationScript_1.mlx is then used to end the propagation and execute a first burn. After the first burn, the satellite enters a transfer ellipse which lasts until the suggested time in IntegrationScript_2.mlx. At that time, the second burn is executed and the satellite is again propagated until it reaches a desired altitude where a second set of maneuvers may be necessary. The script then runs the STK data through the Simulink and Stateflow diagrams found in PAT_2a.slx. The resulting data is output and parsed into a table which tells you when a second set of thrusting maneuvers may be necessary to return the satellite to its intended orbit.

Parameters and Variables

Parameters to be passed from CATIA for this to work are listed below. The script can be easily tested using these variables within the code - ensure that this section is commented out when running the script from the CATIA parametric diagram.

```
%Scenario
    stkVersion = 12;  %string
    scenarioName = 'Grissom';  %string
    scenarioStartTime =  '1 Mar 2023 12:00:00.000'; %string format
    scenarioStopTime = '+1year'; %string format

%Objects
    satelliteName = 'Grissom-2'; %string
    %Ground Station
     facName = 'MC3_AFIT';  %string
     facLatitude = 39.782; %39.782;  %deg
     facLongitude = -84.0832;  %deg
    %Imager Target
     target1Name = 'Riyadh';  %string
     target1Latitude = 24.7136;  %deg
     target1Longitude = 46.6753;  %deg
    %Sensor Target
     target2Name = 'Barcelos';  %string
     target2Latitude = -0.97357;  %deg
     target2Longitude = -62.9269;  %deg
    %Extra Target
%    target3Name = 'Mbandaka';  %string
%    target3Latitude = 0.04865;  %deg
%    target3Longitude = 18.2971;  %deg

%Initial State
```

```
 %Satellite Parameters
     dryMass = 8.765; %kg %7.621kg without propulsion
     Cd = 2.2;
     dragArea = 0.2; %m^2
     fuelMass = 1.235; %kg 1.235
     SRPArea = 0.2; %m^2
     fuelDensity = 0.00000000123; %kg/m^3
     radiationPressureArea = 0.2; %m^2

 %Orbital Elements
      semiMajorAxis = 6828.137;  %km  (altitude is 450 km) (Earth radius
6378.137km)
      eccentricity = 0;  %km
      inclination = 60;  %deg
      argumentPeriapsis = 0;  %deg
      RAAN = 0;  %deg
      trueAnomaly = 0;  %deg

 %Scenario Parameters
 dynamicPropStop1 = 2351280; %s %%Taken from IntegrationScript_1 dynamic
solution
 dynamicPropStop2 = 2357160; %s %%Taken from IntegrationScript_2 dynamic
solution
     stoppingAltitude = 425; %km
     deorbitAltitude = 50; %km

 %Engine Parameters
     IspValue = 40; %s 40
     IspUnit = 's';

     numThruster = 4; %4
     thrust = 25; %mN 25
     thrustUnit = 'mN';


 %C&DH Mask Parameters
     busSOHDataRate = 9600; %bytes/sampletime
     %PayloadSOHDataRate calculated below
     miscDataProductionRate = 0; %ExtraDataRate (bytes)
     storageCapacity = 8; %GB ...converted to bytes for PAT use below
     CDHPowerOn = 2.5; %Watts
     initialMemoryStorageStatus = 1000000000; %bytes  (1GB)

 %EPS Mask Values
     %Battery
     batteryInitialFill = 90; %Percent
```

```
    batteryCapacity = 6; %Ah
    battNominalVoltage = 14.4; %Volts
    chargeDischargeEfficiency = 0.995; %percent
    distroEfficiency = 0.9; %percent, power distribution efficiency
    %Solar Arrays
    PPTefficiency = 0.95; %Peak Power Tracking, percent
    solarArrayArea = 0.1702; %m^2
    solarArrayEfficiency = 0.295; %percent
    %Constraints
    DODtriggerSunSafe = 30; %percent
    DODreturnToNormalOperations = 20; %percent

%ADCS Mask Values
    magnetometerOnPower = 0.825; %Watts
    starTrackerOnPower = 1.8; %Watts
    sunSensorOnPower = 1.35; %Watts
    torqueCoilXOnPower = 0; %Watts
    torqueCoilYOnPower = 0; %Watts
    torqueCoilZOnPower = 0; %Watts

    %Reaction Wheel Values
    reactionWheelMaxPower = 1.925; %Watts
    zeroTorquePower = 0.27; %Watts
    maxMomentum = 0.015; %N-m-s
    reactionWheelMOI = 0.0010577; %kg-m^2
    initialRPM = 0; %rpm
    reactionWheelMaxRPM = 5200; %rpm

%T&C Mask Values
    rxOnPower = 0.7; %Watts
    rxStandbyPower = 0.3; %Watts
    txDataRate = 3200; %DownlinkDataRate (Kbps)
    txOnPower = 8; %Watts

%Payload
     %TIC Imager
    payload1PowerOn = 6.55;  %Watts 6.55
    payload1OnDataRate = 1916.67;  %Bytes/sampletime 1916.67
    payload1StandbyPower = 1.2;  %Watts 1.2
    payload1StandbyDataRate = 0;  %PBytes/sampletime  0
    payload1SOHDataRate = 0;  %Bytes/sampletime 0

    %SensorSDR
    payload2PowerOn = 5;  %Watts 5
    payload2OnDataRate = 1100;  %Bytes/sampletime 1100
    payload2StandbyPower = 2;  %Watts 2
```

```
    payload2StandbyDataRate = 0;  %Bytes/sampletime 0
    payload2SOHDataRate = 0;  %Bytes/sampletime 0

    %Beacon/Comms
    payload3PowerOn = 4; %Watts 4
    payload3OnDataRate = 500; %Bytes/sampletime 500
    payload3StandbyPower = 1.5; %Watts 1.5
    payload3StandbyDataRate = 250; %Bytes/sampletime 250
    payload3SOHDataRate = 0; %Byte/sampletime 0


    %Built in for future use
    payload4PowerOn = 0;  %Watts
    payload4OnDataRate = 0;  %Bytes/sampletime
    payload4StandbyPower = 0;  %Watts
    payload4StandbyDataRate = 0;  %Bytes/sampletime
    payload4SOHDataRate = 0;  %Bytes/sampletime

 %Thruster
    thrusterPowerOn = 12; %Watts
    thrusterHeatOn = 12; %Watts
    thrusterHeatDataRate = 0;
    thrusterOnDataRate = 0;
    thrusterStandbyPower = 1; %Watts
    thrusterStandbyDataRate = 0;
    thrusterSOHDataRate = 0;


    % %Other Variables
     visualizeSimulink = 1;  %toggle for visualizing PAT interface in Simulink
     visualizeResultsFigures = 1;  %toggle visualizing figures
```

Initial Calculations

```
 % Math for data rates and storage inputs from Cameo
 PayloadSOHDataRate = payload1SOHDataRate + payload2SOHDataRate +
payload3SOHDataRate + payload4SOHDataRate;
 storageCapacityConverted = storageCapacity*1000000000; %GB conversion to bytes
 batteryCapWh = batteryCapacity*battNominalVoltage; %Converstion to Wh
 BatteryDataWh = batteryCapWh*(batteryInitialFill/100); %Used in the Battery
Performance diagram

 %Math for orbital calculations
 periapsisRadiusSize = semiMajorAxis*(1-eccentricity); %km
 apoapsisRadiusSize = semiMajorAxis*(1+eccentricity); %km
```

Initiate STK Scenario

```
 CurrentDirectory = pwd;  %defines active directory on user computer where this
script resides

 global TimeStep
 global StartTime
 global EndTime

 %Open STK
     app=actxserver(append('stk',string(stkVersion),'.application'));

    %Grab handle on STK root
     root = app.Personality2;

 %Create New Scenario
     scenario=root.Children.New('eScenario', string(scenarioName));

 %Define Start/Stop time for Scenario
     scenario.SetTimePeriod(string(scenarioStartTime),string(scenarioStopTime));
     root.ExecuteCommand('Animate * Reset');
         %The start time will be reflected in the animation tool bar.
```

Insert the AFIT Ground Station

```
 %Ground Station
     AFIT = scenario.Children.New('eFacility', string(facName));
     AFIT.Position.AssignGeodetic(facLatitude, facLongitude, 297);
```

Other MC3 network ground stations are found below, but will not be used in this thesis.

```
 %Hard-coded MC3 network
 % NPS = scenario.Children.New('eFacility', 'NPS');
 %     NPS.Position.AssignGeodetic(36.597484, -121.873330, 18);
 %
 % SDL = scenario.Children.New('eFacility', 'SDL');
 %     SDL.Position.AssignGeodetic(41.762772, -111.822452, 1401);
 %
 % MLB = scenario.Children.New('eFacility', 'MLB');
 %     MLB.Position.AssignGeodetic(28.021662, -80.679194, 22);
 %
 % HSFL = scenario.Children.New('eFacility', 'HSFL');
 %     HSFL.Position.AssignGeodetic(21.299185, -157.817197, 37);
 %
```

```
% UNM = scenario.Children.New('eFacility', 'UNM');
%     UNM.Position.AssignGeodetic(35.084318, -106.619781, 1584);

% Future MC3 Site Locations
%USNA = scenario.Children.New('eFacility', 'USNA');
    %USNA.Position.AssignGeodetic(38.982065, -76.483940, 11);
%ASF = scenario.Children.New('eFacility', 'ASF');
    %ASF.Position.AssignGeodetic(64.859944, -147.849139, 193);
%CGA = scenario.Children.New('eFacility', 'CGA');
    %CGA.Position.AssignGeodetic(41.375949, -72.101581, 39);
%RSA = scenario.Children.New('eFacility', 'RSA');
    %RSA.Position.AssignGeodetic(34.632413, -86.595497, 170);
%PAC = scenario.Children.New('eFacility', 'PAC');
    %PAC.Position.AssignGeodetic(32.708658, -117.247249, 116);

disp('Ground Stations created...')
```

Insert ground targets

```
%----------Add Target(s)
    % Adds on target
    % Future work - develop a loop that allows user to create a defined
    % number of targets
target1 = scenario.Children.New('eTarget', string(target1Name));
    target1.Position.AssignGeodetic(string(target1Latitude),
string(target1Longitude), 0);
    disp('Target1 created...')

target2 = scenario.Children.New('eTarget', string(target2Name));
    target2.Position.AssignGeodetic(string(target2Latitude),
string(target2Longitude), 0);
    disp('Target2 created...')

% target3 = scenario.Children.New('eTarget', string(target3Name));
%     target3.Position.AssignGeodetic(string(target3Latitude),
string(target3Longitude), 0);
%     disp('Target3 created...')
```

Insert default satellite

Satellite properties will be changed when defining the Initial State

```
%Insert a default satellite
```

```
     satellite=scenario.Children.New('eSatellite',string(satelliteName));
```

Set-up Astrogator

```
 %Change propagator to Astrogator
     satellite.SetPropagatorType('ePropagatorAstrogator')
         % Note that Astrogator satellites by default start with one Initial
State
         % and one Propagate segment

         % Create a handle to the Astrogator portion of the satellites object
model
         % for convenience
     ASTG = satellite.Propagator;
         % Create a handle to the MCS and remove all existing segments
     MCS = ASTG.MainSequence;
     MCS.RemoveAll;
```

Define the Initial State

Included below are the satellite properties of the initial state. To find the properties of your satellite in their initial state, enter the following command into the MATLAB command window: satellite.Propagator.MainSequence.Item(0).InitialState.get

To find how many modules there are and the name of each module, use the following commands:

satellite.Propagator.MainSequence.Count

satellite.Propagator.MainSequence.Item(0).Name

satellite.Propagator.MainSequence.Item(1).Name

etc. for each count...

```
 %Define Initial State
     MCS.Insert('eVASegmentTypeInitialState','Inner Orbit','-');

         % The Insert command will also return a handle to the segment it
creates
     propagate = MCS.Insert('eVASegmentTypePropagate','Propagate','-');

 %Satellite Properties
 satellite.Propagator.MainSequence.Item(0).InitialState.DryMass = dryMass; %kg
based on a 10kg 6U with 0.851kg of propellant
 satellite.Propagator.MainSequence.Item(0).InitialState.Cd = Cd;
```

```
 satellite.Propagator.MainSequence.Item(0).InitialState.DragArea = dragArea;
%m^2
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelMass = fuelMass; %kg
 satellite.Propagator.MainSequence.Item(0).InitialState.FuelDensity =
fuelDensity; %kg/m^3
 satellite.Propagator.MainSequence.Item(0).InitialState.SRPArea = SRPArea; %m^2
 satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureArea =
radiationPressureArea; %m^2
 %satellite.Propagator.MainSequence.Item(0).InitialState.MaxFuelMass=fuelMass;
%kg
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankPressure=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.TankTemperature=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.Cr=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.RadiationPressureCoeff=
;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K1=;
 %satellite.Propagator.MainSequence.Item(0).InitialState.K2=;

 %Change coordinate type to Keplerian
     % Create a handle to the Initial State Segment, set it to use Modified
     % Keplerian elements and assign new initial values
     initstate = MCS.Item('Inner Orbit');
     initstate.OrbitEpoch = scenario.StartTime;
     initstate.SetElementType('eVAElementTypeKeplerian');
     kep=initstate.Element;

  %Define COEs of Initial State
     kep.PeriapsisRadiusSize = periapsisRadiusSize;
     kep.ArgofPeriapsis= argumentPeriapsis;
     kep.Eccentricity= eccentricity;
     kep.Inclination= inclination;
     kep.RAAN= RAAN;
     kep.TrueAnomaly= trueAnomaly;
```

Insert Propagating segment

```
 % Change Propagate segment color
 Red = '0000ff';
 Green = '00ff00';
 Blue = 'ff0000';
 Cyan = 'ffff00';
 Yellow = '00ffff';
 Magenta = 'ff00ff';
 Black = '000000';
 White = 'ffffff';
```

```
 propagate.Properties.Color = uint32(hex2dec(Cyan));

 % Change the propagator type
     propagate.PropagatorName = 'Earth HPOP Default v10';

 %Define stopping condition for propagator
      propagate.StoppingConditions.Add('Duration').Properties.Trip =
dynamicPropStop1; %stops propagating at time specified by the dynamic solution
in part 1
      propagate.StoppingConditions.Remove('Duration'); %removes duration
stopping condition automatically inserted by STK
```

Insert Target Sequence

Create custom engine

Insert target sequence

Define first burn

```
 %-----Create Custom Engine
 %Isp Value conversions
 Ispstr = num2str(IspValue);
 Isp = append(Ispstr, ' ', IspUnit);

 %Thrust Value conversions
  totalThrust = numThruster * thrust;
  totalThruststr = num2str(totalThrust);
  finalThrust = append(totalThruststr, ' ', thrustUnit);

 %Create a new engine model
  root.ExecuteCommand(append('ComponentBrowser */ Duplicate "Engine Models"
"Constant Thrust and Isp" "Cold Gas Thruster"'));
  root.ExecuteCommand(append('ComponentBrowser */ SetValue "Engine Models" "Cold
Gas Thruster" Isp ', Isp ));
  root.ExecuteCommand(append('ComponentBrowser */ SetValue "Engine Models" "Cold
Gas Thruster" Thrust ', finalThrust));

 %-----Insert target sequence (ex: raising the orbit)
      % Insert a Target Sequence with a nested Maneuver segment
         ts = MCS.Insert('eVASegmentTypeTargetSequence','Start Transfer','-');
         dv1 = ts.Segments.Insert('eVASegmentTypeManeuver','DV1','-');
%inserting the first maneuver/first burn
         dv1.Properties.Color = uint32(hex2dec(Red)); %change color in STK

 %-----Define the first burn within the Target Sequence --> Raises orbit
```

```
       dv1.SetManeuverType('eVAManeuverTypeImpulsive'); %sets maneuver to an
impulsive burn
         % Create a handle to the impulsive properties of the maneuver
         impulsive = dv1.Maneuver;
         impulsive.SetAttitudeControlType('eVAAttitudeControlThrustVector');
%attitude control employed in thrust vector
         impulsive.UpdateMass = true; %updates satellite mass as fuel is burned
         impulsive.SetPropulsionMethod('eVAPropulsionMethodEngineModel', 'Cold
Gas Thruster');

         % Create a handle to the Attitude Control - Thrust Vector properties of
the
         % maneuver and set the appropriate axes
         thrustVector = impulsive.AttitudeControl;
         thrustVector.ThrustAxesName = 'Satellite VNC(Earth)';

         dv1.EnableControlParameter('eVAControlManeuverImpulsiveCartesianX');
         dv1.Results.Add('Keplerian Elems/Radius of Apoapsis');

         %Handle to differential corrector profile
         dc=ts.Profiles.Item('Differential Corrector');

         %Set up control parameter

xControlParam=dc.ControlParameters.GetControlByPaths('DV1','ImpulsiveMnvr.Cartes
ian.X');
         xControlParam.Enable=true;
         xControlParam.MaxStep=0.3;

         %Set up result for control parameter
         roaResult = dc.Results.GetResultByPaths('DV1', 'Radius Of Apoapsis');
         roaResult.Enable=true;
         roaResult.DesiredValue = apoapsisRadiusSize;
         roaResult.Tolerance=0.1;

         %Set final differential corrector and targeter properties and run modes
         dc.MaxIterations=50;
         dc.EnableDisplayStatus=true;
         dc.Mode='eVAProfileModeIterate';
         ts.Action='eVATargetSeqActionRunActiveProfiles';
```

Insert Transfer Ellipse

Transfer ellipse will propagate after the first burn until the satellite reaches its desired position for the second burn.

```
    %Propagate the Transfer Orbit to Apogee to prep for second burn
    transferEllipse=MCS.Insert('eVASegmentTypePropagate' , 'Transfer
Ellipse','-');
    transferEllipse.PropagatorName='Earth Point Mass'; %uses the Earth Point
Mass propagator, due to small amount of time spent propagating

    %Add an Stopping Condition and remove the Duration Stopping Condition
    transferEllipseTime = dynamicPropStop2-dynamicPropStop1; %transfer ellipse
will be the amount of time between both burns
    %transferEllipse.StoppingConditions.Add('Apoapsis');
    transferEllipse.StoppingConditions.Add('Duration').Properties.Trip =
transferEllipseTime; %stop propagating at apoapsis
    transferEllipse.StoppingConditions.Remove('Duration'); %remove duration
stopping condition automatically inserted by STK
```

Insert second burn in Target Sequence

This burn re-circularizes the orbit.

```
 % Starting here, we will overwrite some existing variables (ts, dc, etc...)
with a handle to elements in the new target sequence
 ts2 = MCS.Insert('eVASegmentTypeTargetSequence','Finish Transfer','-');
 dv2 = ts2.Segments.Insert('eVASegmentTypeManeuver','DV2','-'); %inserting
second maneuver/burn
 dv2.Properties.Color = uint32(hex2dec(Red));
```

Define the second burn within the Target Sequence

```
 dv2.SetManeuverType('eVAManeuverTypeImpulsive'); %impulsive maneuver
 impulsive = dv2.Maneuver;
 impulsive.SetAttitudeControlType('eVAAttitudeControlThrustVector');
 impulsive.UpdateMass = true;
 impulsive.SetPropulsionMethod('eVAPropulsionMethodEngineModel', 'Cold Gas
Thruster');
 thrustVector = impulsive.AttitudeControl;
 thrustVector.ThrustAxesName = 'Satellite VNC(Earth)';
 dv2.EnableControlParameter('eVAControlManeuverImpulsiveCartesianX');
 dv2.Results.Add('Keplerian Elems/Eccentricity');
```

Set up the Targeter

```
 dc = ts2.Profiles.Item('Differential Corrector');
 xControlParam = dc.ControlParameters.GetControlByPaths('DV2',
'ImpulsiveMnvr.Cartesian.X');
 xControlParam.Enable = true;
```

```
 xControlParam.MaxStep = 0.3;
 eccResult = dc.Results.GetResultByPaths('DV2', 'Eccentricity');
 eccResult.Enable = true;
 eccResult.DesiredValue = eccentricity; %recircularizes the orbit
 eccResult.Tolerance = 0.001;

 % Set final DC and targeter properties and run modes
 dc.EnableDisplayStatus = true;
 dc.Mode = 'eVAProfileModeIterate';
 ts2.Action = 'eVATargetSeqActionRunActiveProfiles';
```

Insert propagating segment for the Outer Orbit

After the second burn, the satellite will begin propagating again. This is the same type of orbit as the initial state propagation. The satellite will continue to orbit until it reaches the stopping altitude, where it is then poised to complete a second set of thrusting maneuvers.

```
 postThrustOrbit = MCS.Insert('eVASegmentTypePropagate','Post Thrust Orbit','-
');
 postThrustOrbit.PropagatorName = 'Earth HPOP Default v10'; %defines propagator
being used
 postThrustOrbit.Properties.Color = uint32(hex2dec(Yellow));
 postThrustOrbit.StoppingConditions.Add('Altitude').Properties.Trip =
stoppingAltitude; %stops propagating at 390 km
 postThrustOrbit.StoppingConditions.Remove('Duration');

 propagate.Results.Add('Time/Duration');
 transferEllipse.Results.Add('Time/Duration');
 postThrustOrbit.Results.Add('Time/Duration');


 ASTG.RunMCS; %runs the entire sequence defined above.

 %Info on mission duration
   disp(['Thrusting End:' num2str(ts2.FinalState.Epoch)]);
   disp(['Scenario End:' num2str(postThrustOrbit.FinalState.Epoch)]);

 %---------Set the Satellite Attitude to slew to Target Area
     % This code slews to Target 1 ONLY
 attitudePointing = satellite.Attitude.Pointing;
 attitudePointing.UseTargetPointing = 1;
 attitudePointing.Targets.RemoveAll;
 attitudePointing.Targets.Add(append('Target/',target1Name));
 attitudePointing.TargetTimes.UseAccessTimes;

 %---------Set the Satellite Attitude to optimal sun-soaking
```

```
 root.ExecuteCommand(append...
     ('SetAttitude */Satellite/',satelliteName,' Profile XPOPInertial Offset
0'));

 %---------Set the Satellite solar panels to fixed position
 root.ExecuteCommand(append...
     ('VO */Satellite/',satelliteName,' InitializeSolarPanelsToSun Enable
Off'));
```

Create Access between Satellite and Target/GS

```
 %-----STK Calculate Ground Station Access
 root.ExecuteCommand(append...
     ('Access */Satellite/',satelliteName,' */Facility/',facName,' TimePeriod
UseScenarioInterval'));

 %-----STK Calculate Target Access
 root.ExecuteCommand(append...
     ('Access */Satellite/',satelliteName,' */Target/',target1Name,' TimePeriod
UseScenarioInterval'));

 root.ExecuteCommand(append...
     ('Access */Satellite/',satelliteName,' */Target/',target2Name,' TimePeriod
UseScenarioInterval'));

 disp('Access established between Satellite with Target1, Target2 and Ground
Station...')

 %-----Was the scenario created?
     % Counts if there is an open STK scenario and sends a boolean response

 exeName = 'AgUiApplication';

 [status,result] = system('tasklist/FI "imagename eq AgUiApplication.exe"');

 num = count(result, exeName);

 if num >= 1
     Create_Scenario = true;
 else
     Create_Scenario = false;
 end

 disp('Scenario created...')
```

Report Generation

Generates reports needed to run PAT Access (target and ground), Attitude, Ephemeris, Lunar, and Sun Units for scenario set to Epoch Seconds for all reports.

```
 %-----Set the report date/time format for all exported data files:
 root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');
 root.ExecuteCommand('Units_Set * All Date EpSec ConnectReportUnitsFlag On');

 disp('Epsec Units set for PAT')


 %-----Generate LLA Report for Satellite to later determine altitude
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "LLA
Position" File "',CurrentDirectory,'\LLAPosition.csv" TimePeriod
UseScenarioInterval'));

 %-----Generate Access Report for Satellite to ground station
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportGS.csv" AccessObject
*/Facility/',facName,''));

 disp('Access to Ground Station Report Created...')

 %-----Generate Gap Report for Satellite to ground station
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsGS.csv" AccessObject */Facility/',facName,''));

 disp('Access Gaps to Ground Station Report Created...')

 %-----Generate Access Report for Satellite to Target(s)
 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget1.csv" AccessObject
*/Target/',target1Name,''));

 root.ExecuteCommand(append...
     ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Access"
File "',CurrentDirectory,'\AccessReportTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access to Target(s) Report Created...')

 %-----Generate Gap Report for Satellite to Target(s)
 root.ExecuteCommand(append...
```

```
    ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget1.csv" AccessObject
*/Target/',target1Name,''));

 root.ExecuteCommand(append...
    ('ReportCreate */Satellite/',satelliteName,' Type Export Style "Gaps" File
"', CurrentDirectory,'\GapsTarget2.csv" AccessObject
*/Target/',target2Name,''));

 disp('Access Gaps to Target(s) Report Created...')

 %-----Generate Ephemeris file
 root.ExecuteCommand(append...
    ('ExportDataFile */Satellite/',satelliteName,' Ephemeris
"',CurrentDirectory,'\Satellite1.e" Type STK CoordSys J2000 TimeSteps 60
CentralBody Earth'));

 disp('Ephemeris Report Created...')

 %-----Generate Attitude file
 root.ExecuteCommand(append...
    ('ExportDataFile */Satellite/',satelliteName,' Attitude
"',CurrentDirectory,'\Satellite1.a" CoordAxes J2000 TimeSteps 60 '));

 disp('Attitude Report Created...')

 %-----Generate Moon Data .txt file
 root.ExecuteCommand(append...
    ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Lunar Vector
J2000" File "',CurrentDirectory,'\moon.txt" TimePeriod UseScenarioInterval'));

 disp('Moon Report Created...')

 %-----Generate Sun Data .txt file
 root.ExecuteCommand(append...
    ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Sun Vector
J2000" File "',CurrentDirectory,'\sun.txt" TimePeriod UseScenarioInterval'));

 disp('Sun Report Created...')

 %-----Generate Maneuver Data .txt file
 root.ExecuteCommand(append...
    ('ReportCreate */Satellite/',satelliteName,' Type Save Style "Maneuver
Summary" File "',CurrentDirectory,'\maneuver.txt"'));
 %Maneuver Summary only includes DeltaV and Fuel Used, it does not include
 %the semimajor axis information, that is why you need the LLAPosition data.
```

```
disp('Maneuver Report Created...')
```

MAT FILE CONVERSIONS FOR PAT

Convert Moon and Sun Files

Convert Attitude and Ephemeris Files

txt and cvs files converted to .mat files and saved in current dir.

```
%-----Load Moon File
    fprintf(1,'Lunar Position...          ')
    moon = (readmatrix('moon.txt'))';
    save moon.mat moon
        fprintf('Done\n');

%-----Load Sun File
    fprintf(1,'Solar Position...           ');
    sun = (readmatrix('sun.txt'))';
    save sun.mat sun
            fprintf('Done\n');

%-----Load Attitude and Ephemeris File
file_att = 'Satellite1.a';
file_eph = 'Satellite1.e';

data_att = readData(file_att);
save att.mat data_att
ephem = readData(file_eph);
save ephem.mat ephem

%------Load LLA Position File
LLA = (readmatrix('LLAPosition.csv'))';
save LLA.mat LLA

%-----Load Maneuver File
    fprintf(1,'Maneuver...                  ');
    maneuver = (readmatrix('maneuver.txt'))';
    maneuver(isnan(maneuver(:,1)),:) = [];
    save maneuver.mat maneuver
            fprintf('Done\n');

%------Load .mat files to Workspace
    load('moon.mat');
    load('sun.mat');
    load('maneuver.mat');
```

```
    load('att.mat');
    load('ephem.mat');
    load('LLA.mat');
```

READ AND PARSE ACCESS REPORTS FOR GROUND STATION

Reads Ground Station Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Ground Station access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessGS = readmatrix('AccessReportGS.csv');
accessGS(isnan(accessGS(:,1)),:) = [];
%accessGS(isnan(accessGS)) = []; %use if there is an error when running this
script --> accessGS(isnan(accessGS(:,1)),:) = [];

% Using moon.mat's format, determine time-related variables:
TimeStep = moon(1,2) - moon(1,1);
StartTime = moon(1,1);
EndTime = moon(1,end-1);

% Logic check to verify that TimeStep is consistent throughout entire moon.mat
array
DiffTimeStep = max(diff(moon(1,end-1))) - min(diff(moon(1,end-1))); %If the
last TimeStep is included, the TimeStep will not be equal to 60 seconds. The
scenario time is not a perfect dividen of 60.
    if DiffTimeStep ~= 0
        fprintf('CAUTION: TimeStep may not be of equal intervals \n');
    else
        fprintf('TimeStep is...         %d seconds \n',TimeStep);
    end

% Using time-related variables, create a 1-by-X time-vector (tv)
tv = (StartTime:TimeStep:EndTime);

% Pre-allocate memory for contact vector (contact)
contact = zeros(size(tv));

% Systematically parse the Access Report and update the contact vector to
% have a "1" for durations of time when the satellite is in view of the GS
for k = 1:size(accessGS,1)
    inview = find(tv > accessGS(k,2) & tv < accessGS(k,3));
    contact(inview) = 1;
```

```
end

% Finally, combine time and contact vectors to be used by CSAT
InViewofGS = transpose([tv; contact]);
```

READ AND PARSE ACCESS REPORTS FOR TARGET

Reads Target Access Reports

Sets timestep

Creates time and contact vectors for CSAT use

```
% Read Target(s) access report & remove any NaN/blank rows
% Read structure: # | start time | stop time | duration
accessTarget1 = readmatrix('AccessReportTarget1.csv');
accessTarget1(isnan(accessTarget1(:,1)),:) = [];
%accessTarget1(isnan(accessTarget1)) = []; %error when running this script -->
accessTarget1(isnan(accessTarget1(:,1)),:) = [];

accessTarget2 = readmatrix('AccessReportTarget2.csv');
accessTarget2(isnan(accessTarget2(:,1)),:) = [];
%accessTarget2(isnan(accessTarget2)) = []; %error when running this script -->
accessTarget2(isnan(accessTarget2(:,1)),:) = [];

% Pre-allocate memory for capture vector (capture)
capture1 = zeros(size(tv));
capture2 = zeros(size(tv));
```

Systematically parse the Access Report and update the capture vector to have a "1" for durations of time when the satellite is in view of the target.

However, unlike the contact vector, the payload turns on when it is most nadir (i.e. directly overhead) to the target. CSAT logic determines when to enter Standby mode and On mode.

REFERENCE: script was first developed by Lt Heather Udell (see timeON.m script)

```
for z = 1:size(accessTarget1,1)

    ti = accessTarget1(z,2);    % Time it enters field of view
    tf = accessTarget1(z,3);    % Time it exits field of view
    DeltaT = (tf - ti);        % Duration of pass
    nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

    % The "nadir" time needs to be rounded to the closest second
    % that is a multiple of the scenario's timestep
```

```
    nadirTimeStep = TimeStep*(round(nadir/TimeStep));

    payloadON = find(tv == nadirTimeStep);
    capture1(payloadON) = 1;

 end

  for z = 1:size(accessTarget2,1)

    ti = accessTarget2(z,2);    % Time it enters field of view
    tf = accessTarget2(z,3);    % Time it exits field of view
    DeltaT = (tf - ti);        % Duration of pass
    nadir = ti + (DeltaT/2);   % Time the CubeSat is directly overhead

    % The "nadir" time needs to be rounded to the closest second
    % that is a multiple of the scenario's timestep
    nadirTimeStep = TimeStep*(round(nadir/TimeStep));

    payloadON = find(tv == nadirTimeStep);
    capture2(payloadON) = 1;

  end

% Finally, combine time and capture vectors to be used by CSAT
InViewofTarget1 = transpose([tv; capture1]);
InViewofTarget2 = transpose([tv; capture2]);
```

READ AND PARSE LLA POSITION

```
LLAPosition = readmatrix('LLAPosition.csv');
Altitude = LLAPosition(2:end,4);
```

READ AND PARSE THRUSTING DATA

Creates a data set to tell CSAT when thrusting occurs for power draw.

Uses time step created above.

```
% Pre-allocate memory for contact vector (contact)
thrustTime = zeros(size(tv));
```

Systematically update the thrusting vector to have a "2" for durations of time when the satellite propulsion system in heating and a "1" for durations of time when the satellite propulsion system is thrusting. The heating duration is set to 1800 seconds generically, but should be modified to be specific to the heating necessary for the specific propulsion system being used.

```
 heatingDuration = 1800; %s
 heatingStart = dynamicPropStop1 - heatingDuration;
 heatingEnd = dynamicPropStop1 - 60;

 heatingRow1 = dynamicPropStop1;

 for k = 1:size(tv,1)
     heating1 = find(tv < dynamicPropStop1 & tv > dynamicPropStop1 -
heatingDuration);
     heating2 = find(tv < dynamicPropStop2 & tv > dynamicPropStop2 -
heatingDuration);
     heating = [heating1 ; heating2];
     thrustTime(heating) = 2;

     thrustTime1 = maneuver(5,1);
     thrustTime2 = maneuver(5,2);
     thrusting = find((tv >= dynamicPropStop1 & tv < dynamicPropStop1 +
thrustTime1) | (tv >= dynamicPropStop2 & tv < dynamicPropStop2 + thrustTime2) );
     thrustTime(thrusting) = 1;
 end

 % Finally, combine time and thrusting vectors to be used by CSAT
 Thrusting = transpose([tv; thrustTime]);
```

CREATE SPACE FOR WORKSPACE TASK LIST GENERATED FROM CSAT

```
     % Pre-allocate memory for Workspace Task List
         WSTaskList = zeros(length(tv),15);
```

RUN THE PAT

```
     % Runs the PAT with the input values from the workspsace
     % Opens PAT for visualization (can turn on/off)

 if visualizeSimulink == 1
     open_system('PAT_2a');
 end

 sim(['PAT_2a' ...
     '']);
```

MAKEFIGURES AND DISCRETE OUTPUTS

Outputs the makefigures.m plots

Max/Min Bus Voltage

Max/Min Solar Array Power Generation

Battery Depth of Discharge

Reaction Wheel Speed (any direction)

Max/Min Torque (any direction)

Buffer Capacity Status

```
%Plot Telemetry Data
if visualizeResultsFigures == 1
    makefigures;
end

%SA Power Generation
minSAPowerGen = min(Telem(:,20));  %Watts %This is producing the minimum Solar
Array Power Generation
maxSAPowerGen = max(Telem(:,7));  %Watts

%Total Component Power
minPowerTot = min(Telem(:,21)) %Watts %This is producing the minimum total
component power
maxPowerTot = max(Telem(:,21)) %Watts

%Battery Depth of Discharge
maxDoD = max(Telem(:,6));  %percent
save DoD.mat t DoD

%Reaction Wheel Speed in any direction
maxRPM = max([max(Telem(:,14)), max(Telem(:,15)), max(Telem(:,16))]);  %RPM

%Max and Min Torque in any direction
maxTorque = max([max(Telem(:,17)), max(Telem(:,18)), max(Telem(:,19))]);  %N-m
minTorque = min([min(Telem(:,17)), min(Telem(:,18)), min(Telem(:,19))]);  %N-m

%Buffer (Memory Space)
maxBuffer = max(Telem(:,4));  %Percent Full

%Thruster
%Convert maneuver.mat to maneuver.csv
maneuvermat = load('maneuver.mat');
csvwrite('maneuvercsv.csv', maneuvermat.maneuver);
load('maneuvercsv.csv');
 fuelUsed = maneuvercsv(3,8); %kg
```

```
totDeltaV = maneuvercsv(3,7); %m/s
scenarioEnd = num2str(postThrustOrbit.FinalState.Epoch);
```

FAULT MODE OUTPUTS

If the fault mode was activated during scenario, the output is true

If the fault mode was not activated during scenario, the output is false

```
%RWALimit Fault Mode
RWALimit = FaultModeStatus(:,1);
if RWALimit == 0
    RWALimitStatus = false;  %no fault mode triggered
else
    RWALimitStatus = true;  %fault mode triggered
end

%Sun Safe Fault Mode
SunSafe = FaultModeStatus(:,2);
if SunSafe == 0
    SunSafeStatus = false;  %no fault mode triggered
else
    SunSafeStatus = true;  %fault mode triggered
end

%Survival Status Fault Mode
Survival = FaultModeStatus(:,3);
if Survival == 0
    SurvivalStatus = false;  %no fault mode triggered
else
    SurvivalStatus = true;  %fault mode triggered
end

%Memory Limit Fault Mode
MemoryLimit = FaultModeStatus(:,4);
if MemoryLimit == 0
    MemoryLimitStatus = false;  %no fault mode triggered
else
    MemoryLimitStatus = true;  %fault mode triggered
end
```

QUASI-STATIC SOLUTION

The quasi-static solution finds a time when a second set of thrusting maneuvers would be necessary based on where all of the Sol criteria are true:

- Depth of Discharge < 20
- Altitude < 430
- Not accessing a ground station or target

```
 accessGS = InViewofGS(:,2);
 accessT1 = InViewofTarget1(:,2);
 accessT2 = InViewofTarget2(:,2);
 DynamicData = table(t,DoD,Altitude,accessGS,accessT1,accessT2);

 Sol = (DynamicData.t>dynamicPropStop2 & DynamicData.DoD<20 &
DynamicData.Altitude<430 & DynamicData.accessGS==0 & DynamicData.accessT1==0 &
DynamicData.accessT2==0);
 DynamicData(Sol,:)

 %The DynamicData table gives you the solution for a time that you should
 %begin the next thrusting maneuver to continue your mission.
```

Attitude and Ephemeris Function to create ephem.mat and att.mat

Developed by David Evert

```
 function data = readData(filename)

     fprintf('Reading data from %s...\n', filename);

     % File format based on file extension
     [~,~,extn] = fileparts(filename);
     if strcmp(extn,'.a')
         key = 'AttitudeTimeQuaternions';    % Key word that comes before data
         nskip = 0;                          % Number of lines between key word
and data
         format = '%f %f %f %f %f';          % Format of data
         nCols = 5;                          % Number of columns in data
     elseif strcmp(extn,'.e')
         key = 'EphemerisTimePosVel';        % Key word that comes before data
         nskip = 1;                          % Number of lines between key word
and data
         format = '%f %f %f %f %f %f %f';    % Format of data
         nCols = 7;                          % Number of columns in data
     end
```

```
    % Open file for reading
    fid = fopen(filename,'r+');

    % Read number of lines of data
    % Loop through file lines until it gets to NumberOf... line
    try
        flag = true;
        while flag
            currentLine = fgets(fid);
            if contains(currentLine,'NumberOf')
                flag = false;
                % Find number in current line and save to variable
                lines = str2double(currentLine(regexp(currentLine, '[\d]')));
            end
        end
    catch
        fprintf('File does not contain NumberOf... value\n');
    end

    % Find line containing keyword, then skip nskip lines after it
    try
        flag = true;
        while flag
            if contains(fgets(fid),key)
                flag = false;
                for n = 1:nskip
                    fgets(fid);
                end
            end
        end
    catch
        fprintf('File does not contain "%s" key word\n', key);
    end

    % Read data and save to .mat file
    data = fscanf(fid, format, [nCols,lines]);

    % Close file
    fclose(fid);

    fprintf('Done\n');
end
```

Data Provider Function

If additional data providers are needed for your mission, use this data provider function. It makes it very easy to pull information from STK.

```
 function [dataFull] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect)

 %Analytical Graphics, Inc.
 %Author: Mo Syed
 %Date Created: 9/12/18
 %Edit 3/11/20

 %This function takes in the data provider parameters and outputs the
 %desired data, skipping the setup that is usually needed. It automates the
 %process without the user having to get into the semantics of how data
 %providers work in object model.

 %It is also useful to have the report & graph manager open as if you were
 %trying to create a custom report so you can see the possible data provider
 %and proper subfolders to use as function inputs

 %root is the STK root.

 %dataProvString is the data provider from STK.

 %times corresponds to the start, stop and step time (put in values
 %accordingly). Pass in as a cell array.

 %dataProvElem is the actual elements of data you are looking for. Pass in
 %as a cell array. If you leave this as empty in the input (as a []), the
 %function will pull ALL of the data elements in the data provider. They
 %will not have the name of the data elements above them but they correspond
 %to the order found in the report & graph manager in STK.

 %grouping is the subfolder under the data provider type in STK. Not every
 %data provider will have this so leave empty (as a []) in the inputs if
 %that is the case. Pass it in as a string

 %predata is the input of predata in case your dataprovider needs predata.
 %Leave this empty (as a []) in the inputs if no predata is needed.

 %object is the STK object the data provider is on. Pass on the handle to the
 %object

 %--------SPECIFIC INSTRUCTIONS FOR DATA PROVIDERS THAT NEED PREDATA--------
```

```
%You can enter in predata one of two ways into the script. The first way is
%to manually know your predata string and place that into the predata input
%listed above. The second is to allow a UI window to open up in STK that
%allows you to select the predata as you run this function. The predata
%input method allows you to keep the code streamlined, while having the UI
%window open up makes it easier for the user to select the correct predata
%with less room for error and figuring out what predata is needed. If you
%don't know if your data provider has predata, enable the option so the
%ui window will automatically popup even if your predata input is empty
%(as a [])


%To enable the noncoding UI option for predata, set the value of
%UIPredata = 1 below. If you want it off, leave it as UIPredata = 0.


%----------------Example of variable inputs for the function---------------

%     root is your handle to Personality2
%     dataProvString = 'Axes Choose Axes';
%     dataProvElem = {'Time','q1','q2','q3','q4'};
%     times = {0,2400,60};
%     grouping = 'Body';
%     predata = [CentralBodies/Earth];
%     object = root.GetObjectFromPath('Satellite/Satellite1');
%     [outputData] =
DataProviderFunction(root,dataProvString,dataProvElem,times,grouping,predata,obj
ect);


%---------------------------------Code-----------------------------------

    %Grabs this to convert units properly
    scenario = root.CurrentScenario;

    %Refer to the predata readme section

    UIPredata = 0;

    %Grabs universal data provider interfaces

    dataProviderValue = object.DataProviders.Item(dataProvString);
    dataProviderType = dataProviderValue.Type;


    %Determines if time is in UTCG or EpSec
```

```
    if strcmp(class(times{1}),'double') %Epsec are double inputs
        root.UnitPreferences.Item('DateFormat').SetCurrentUnit('EpSec');
    elseif strcmp(class(times{1}),'char') %UTCG is char inputs
        root.UnitPreferences.Item('DateFormat').SetCurrentUnit('UTCG');
    end
    %Determines if there is a subfolder on the dataprovider
    if dataProviderValue.IsGroup == 0
        objectDP = dataProviderValue;
    elseif dataProviderValue.IsGroup == 1
        try
            objectDP = dataProviderValue.Group.Item(grouping);
        catch
            error(['This data provider has a grouping input that needs to'...
                ' be assigned. Look at the report & graph manager in STK'...
                ' for the proper subfolder in the data provider that you
are'...
                ' trying to extract data for.'])
        end
    end

    %Determines if there is predata included
    objectDP.AllowUI = UIPredata;
    if ~isempty(predata)
        if UIPredata == 0
            objectDP.PreData = predata;
```

```
        end

    end


    %This is where things differ depending on report type

    switch (dataProviderType)

        case 'eDrTimeVar'

            dataProviderFinal = objectDP.Exec(times{1},times{2},times{3});

        case 'eDrIntvl'

           dataProviderFinal = objectDP.Exec(times{1},times{2});

        case 'eDrFixed'

           dataProviderFinal = objectDP.Exec;

    end

    %Determines if you want all data elements or a specified list

    dataCell = [];

     if ~isempty(dataProvElem)

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1



            for i = 1:length(dataProvElem)

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.GetDataSetByName(dataP
rovElem{i}).GetValues;
                    dataCell = [dataCell; data];
```

```
                end

                dataElem(:,i) = dataCell;
                dataCell = [];
            end

        elseif dataProviderFinal.Interval.Count == 1

            for i = 1:length(dataProvElem)

                dataElem{i} =
dataProviderFinal.DataSets.GetDataSetByName(dataProvElem{i}).GetValues;

            end

        else

                dataElem = {['No ' dataProvString ' Data Available']};

        end

      else

        %Checks if object and if it has more than one data interval. For
        %example, access data have more than one data set for it that
        %corresponds to each access interval

        if dataProviderFinal.Interval.Count > 1

                for j = 0:dataProviderFinal.Interval.Count-1

                    data =
dataProviderFinal.Interval.Item(cast(j,'int32')).DataSets.ToArray();
                    dataCell = [dataCell;  data];

                end

                dataElem = dataCell;
                dataCell = [];

        elseif dataProviderFinal.Interval.Count == 1

                dataElem = dataProviderFinal.DataSets.ToArray();

        else
```

```
                dataElem = {['No ' dataProvString ' Data Available']};

        end

     end


    for i = 1:length(dataProvElem)

        dataColumn = [dataProvElem{i} ; dataElem{i}];
        dataFull(:,i) = dataColumn;

    end



end
```

Bibliography

[1] K. Henderson & A. Salado, "Value and benefits of model-based systems engineering (MBSE): Evidence from the literature," *Systems Engineering,* vol. 24, no. 1, pp. 51-66, 2021.

[2] J. Duprez, "An MBSE modeling approach to efficiently address complex systems and scalability," *INCOSE International Symposium,* vol. 28, no. 1, pp. 940-954, 2018.

[3] U.S. Department of Defense, "Defense Space Strategy Fact Sheet," 2020.

[4] A. Wilson, "Acquisition Chief Calls for Disruptive Agility," *New Digital Paradigm,* 2020.

[5] United States, "U.S. national space policy," Office of Science and Technology Policy, Washington, D.C., 2020.

[6] D. Kaslow & A. M. Madni, "Validation and verification of MBSE-compliant CubeSat reference model," *Disciplinary Convergence in Systems Engineering Research,* pp. 381-393, 2018.

[7] D. Kaslow, B. Ayres, P. T. Cahill, L. Hart & R. Yntema, "A Model-Based Systems Engineering (MBSE) approach for defining the behaviors of CubeSats," *IEEE Aerospace Conference,* pp. 1-14, 2017.

[8] A. Rasheed, O. San and T. Kvamsdal, "Digital twin: Values, challenges and enables from a modeling perspective," *IEEE Access,* vol. 8, 2020.

[9] C. Zhuang, J. Gong and J. Liu, "Digital twin-based assembly data management and process traceability for complex products," *Journal of manufacturing systems,* vol. 58, pp. 118-131, 2021.

[10] U.S. Air Force, "Space Test Program," 2007.

[11] AFIT CSRA, "Grissom-2 SERB," 2020.

[12] S. Friedenthal, A. Moore and R. Steiner, A Practical Guide to SysML, Waltham, MA: Elsevier Inc., 2015.

[13] A. M. Madni and M. Sievers, "Model-based systems engineering: Motivation, current status, and research opportunities," *Wiley,* vol. 21, pp. 172-190, 2018.

[14] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*, Pearson Education, Inc., 2014.

[15] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," *INCOSE MBSE Focus Group,* vol. 25, no. 8, pp. 1-12, 2007.

[16] K. K. Brown, "Logic-Based Mission Modeling Tool Evolution For CubeSat Payload Analysis Using Model-Based Systems Engineering," MS Thesis, Air Force Institute of Technology, 2021.

[17] "SysML Tools," [Online]. Available: https://sysmltools.com/.

[18] J. Puig-Suari, C. Turner and W. Ahlgren, "Development of the Standard CubeSat Deployer and the CubeSat Class PicoSatellite," *2001 IEEE aerospace conference proceedings,* vol. 1, pp. 1-347, 2001.

[19] A. Toorian, K. Diaz and S. Lee, "The CubeSat Approach to Space Access," in *2008 IEEE Aerospace Conference,* 2008.

[20] C. Cappelletti, S. Battistini and B. Malphrus, CubeSat Handbook: From Mission Design to Operations, Elsevier Inc. , 2021.

[21] A. Poghosyan and A. Golkar, "CubeSat evolution: Analyzing CubeSat capabilities for conducting science missions," *Progress in Aerospace Sciences,* vol. 88, pp. 59-83, 2017.

[22] Cal Poly SOL The CubeSat Program, "6U CubeSat Design Specification Revision 1.0," 2018.

[23] Planetary Systems Corporation, "Payload Specification for 3U, 6U and 12U," 6 August 2018. [Online]. Available: https://planetarysystemscorp.com/wp-content/uploads/2002367F-Payload-Spec-for-3U-6U-12U.pdf. [Accessed April 2022].

[24] NASA, "State-of-the-Art Small Spacecraft Technology," National Aeronautics and Space Administration , Moffett Field, CA, 2020.

[25] A. R. Tummala and A. Dutta, "An Overview of Cube-Satellite Propulsion Technologies and Trends," *Aerospace,* vol. 4, no. 58, 2017.

[26] M. Zaberchik, D. R. Lev, E. Edlerman and A. Kaidar, "Fabrication and Testing of the Cold Gas Propulsion System Flight Unit for the Adelis-SAMSON Nano-Satellites," *Aerospace,* vol. 6, no. 91, 2019.

[27] J. R. Wertz, D. F. Everette and J. J. Puschell, Space Mission Engineering: The New SMAD, Torrance, CA: Microcosm Press, 2018.

[28] K. Lemmer, "Propulsion for CubeSats," *Acta Astronautica,* vol. 134, pp. 231-243, 2017.

[29] D. Kaslow, G. Soremekun, H. Kim and S. Spangelo, "Integrated Model-Based Systems Engineering (MBSE) Applied to the Simulation of a CubeSat Mission," in *2014 IEEE Aerospace Conference*, 2014.

[30] J. McCarty, "A Simulink Based Tool for Design Reference Mission Modeling," Air Force Institute of Technology, 2010.

[31] B. A. Andrews, "A Colony II CubeSat Mission Modeling Tool," Air Force Institute of Technology, 2012.

[32] A. Hatch, "Electrospray Propulsion Interface and Mission Modeling for CubeSats," MS Thesis, Air Force Institute of Technology, 2012.

[33] H. M. Udell, "A CubeSat Mission Modeling Tool," MS Thesis, Air Force Institute of Technology, 2015.

[34] B. A. Jewell, "Applying Model-Based Systems Engineering to CubeSats: A Tailored Approach For a Reusable State Analysis Tool," MS Thesis, Air Force Institute of Technology, 2015.

[35] J. R. Loudermilk, "A Logic-Based Mission Modeling Tool for Designing CubeSats," MS Thesis, Air Force Institute of Technology, 2016.

[36] J. Sadowski, "Dynamic Logical Mission Modeling Tool," MS Thesis, Air Force Institute of Technology, 2017.

[37] S. Kelly, "A Reference Architecture for Rapid CubeSat Development," MS Thesis, Air Force Institute of Technology , 2021.

[38] S. T. Luther, "SysML Based CubeSat Model Design and Integration With The Horizon Simulation Framework," California Polytechnic State University, 2016.

[39] A. Johnson, "CubeSat Astronomy Mission Modeling Using the Horizon Simulation Framework," California Polytechnic State University, 2019.

[40] A. G. Incorporated, "Moxie Spec Sheet," 2021. [Online]. Available: https://www.agi.com/getmedia/6ed61973-eb43-44d5-bb10-111b6111f7f5/Moxie-sheet.pdf.

[41] N. D. Erbe and D. P. Lemmer, "Applying Model-Based Systems Engineering and Fidelity Quantification to Support Fair Fight in a Distributed Simulation System," Air Force Institute of Technology, 2021.

[42] N. Saeed, E. A. H. Ahmed, H. Dahrouj, T. Y. Al-Naffouri and M.-S. Alouini, "CubeSat Communications: Recent Advances and Future Challenges," *IEEE Communications Surveys and Tutorials,* vol. 22, no. 3, pp. 1839-1862, 2020.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
06/16/2022	Master's Thesis	August 2020 - June 2022

4. TITLE AND SUBTITLE
A Model-Based Approach to CubeSat Propulsion and Payload Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Johnson, Madeline H., Capt, USSF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENV-MS-22-J-063

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Space Warfighting Analysis Center
Dr. George Boyarko
7250 Getting Heights
Colorado Springs, CO 80916
george.boyarko@spaceforce.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
Currently, there are only limited ways to increase mission success of CubeSats in terms of component and mission compatibility. The Payload Analysis Tool (PAT), developed by Air Force Institute of Technology (AFIT) students, combines the power of multiple tools to analyze payload compatibility on a single CubeSat bus. The PAT simulates a CubeSat mission with a variety of payloads to better understand how the payloads interact with the bus in terms of power, data rate, and memory, but it lacks a propulsion system. This thesis research advances the PAT by including a propulsion system which allows for increased mission time and unearths the issues CubeSats face after being in space for an extended period.

15. SUBJECT TERMS
Model-Based Systems Engineering, CubeSat, Propulsion, Mission Modeling Tool

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. David R. Jacques, AFIT/ENV
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	207	(937) 255-6565; david.jacques@afit.edu