SYSML OUTPUT INTERFACE AND SYSTEM-LEVEL REQUIREMENT ANALYZER FOR THE HORIZON SIMULATION FRAMEWORK

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

By

Viren Kishor Patel

April 2018

COMMITTEE MEMBERSHIP

TITLE:	SysML Output Interface and System-Level Requirement Analyzer for the Horizon Simulation Framework
AUTHOR:	Viren Kishor Patel
DATE SUBMITTED:	April 2018
COMMITTEE CHAIR:	Dr. Eric Mehiel, Ph. D. Department Chair, Professor of Aerospace Engineering
COMMITTEE MEMBER:	Dr. Jordi Puig-Suari, Ph. D. Professor of Aerospace Engineering
COMMITTEE MEMBER:	Dr. Kira Abercromby, Ph. D. Associate Professor of Aerospace Engineering
COMMITTEE MEMBER:	Dr. Kurt Colvin, Ph. D. Professor of Industrial and Manufacturing Engineering

ABSTRACT

SysML Output Interface and System-Level Requirement Analyzer for the Horizon Simulation Framework

Viren Kishor Patel

Model-Based Systems Engineering in industry has been constantly increasing its presence within the aerospace industry. SysML is one such MBSE tool that shows complex system organization and relationships. The Horizon Simulation Framework is another MBSE tool, created by Cal Poly students, that gives users the ability to run "day-in-the-life" simulations of systems. Finding a way to link these two tools could allow systems engineers to reap the benefits of both.

This thesis investigates the background and design process involved with developing the code that can convert an output file generated in SysML, into a format specifically made for the Horizon Simulation Framework. The goal was to create an interface that can allow users to model a system in SysML, and analyze the model and verify system requirements using HSF. Another goal was to expand the capabilities of the Horizon Simulation Framework by designing and develop a module that would allow users to define and analyze system-level requirements. To evaluate the effectiveness of both codes, the Aeolus example case was used. A SysML model of the system was created as the product of another thesis; *SysML based CubeSat Model Design and Integration with the Horizon Simulation Framework*. The Aeolus SysML model was converted and used as input in an HSF simulation. The SysML model simulation data was compared against those of the original test case. To test the requirement module, system level requirements were formulated within the Aeolus system and run in simulation, providing an analysis of the results. The results of the analysis confirmed a successful conversion of the SysML model into an equivalent HSF model and a successful analysis of system-level requirements.

ACKNOWLEDGMENTS

This thesis is dedicated to my family. My parents, Kishor and Gita Patel, taught me to never give up in the face of adversity and that no matter where I go, they're always there for me. My sister, Vanisha Patel, taught me that I'm not alone in this world. My grandmother, Godavariben Patel, taught me with great patience comes great rewards. My loving wife, Ruchi Patel, taught me that life is too short to worry. I'm thankful for all of them and I have no idea where I would be without them. I love you all.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF ACRONYMS

BDD:	Block Definition Diagram
Cal Poly:	California Polytechnic State University, San Luis Obispo
ConOps:	Concept of Operations
EOM:	Equations of Motion
HSF:	Horizon Simulation Framework
IBD:	Internal Block Diagram
LINQ:	Language Integrated Query
MBSE:	Model-Based Systems Engineering
MDO:	Multi-Disciplinary Optimization
PLM:	Product Life Management
SE:	Systems Engineering
SoS:	System of Systems
SSDR:	Solid State Data Recorder
STK:	System Tool Kit
SysML:	Systems Modeling Language
UML:	Unified Modeling Language
XML:	Extensible Markup Language

INTRODUCTION

1.1 Problem Statement and Proposition

Engineered systems are ever rapidly increasing in complexity. This increased complexity has spawned a demand for tools that can model these multi-faceted systems. The demand for such tools has arisen from the growing study of model-based systems engineering, or MBSE. These new modeling tools have empowered systems engineers to represent and develop systems in countless ways for every step of the design and build life cycle.

Despite the advent of new and advanced modeling tools, the fundamental concepts of systems engineering still remain. Elements like system validation are still an integral step in ensuring that the system operates per established specifications. The requirements verification process has even become even more crucial to prove that a system can perform within given constraints and satisfy customer needs. Implementation of document-based systems engineering alone cannot keep up the demands of today's ever-increasingly complex systems.

One prominent MBSE resource, Systems Modeling Language, or SysML, has proven to be a powerful tool for modeling complex systems in the aerospace industry. SysML users can visualize a system and its components while specifying features, requirements, relationships, and actions. Even with its myriad of capabilities, SysML is lacking in simulating system behavior and verifying requirements. The Cal Poly-created Horizon Simulation Framework is another MBSE tool used to model systems, but its specialty differs from SysML. HSF can simulate modeled systems through existing, student-created software. However, HSF can expand its capabilities to work with such programs as SysML to increase its functionality. Creating a channel of connectivity from SysML to the Horizon Simulation Framework could allow engineers to model systems without being pigeon-holed into using a single tool. These new model-based systems engineering tools stand to benefit from one another, resulting in a call for interconnectivity.

HSF lacks in the ability to analyze systems as a whole. SysML cannot grant HSF this proficiency, but furnishing HSF with the power to investigate operations at the system level

would better balance the program. Currently, HSF individually looks at each of these parts of a system individually. What HSF needs is a way to verify requirements that concern the entire system. Doing so would allow HSF to provide more realistic results to its users.

The focus of this thesis centers on two major sections of HSF code. The first section entails converting systems modeled in SysML into a file format readable by the Horizon Simulation Framework. HSF will then be able to use this reformatted file to verify that that the modeled system meets the stated objectives. This code will act as the link between system modeling and simulation. The second section of code will analyze user-defined system-level requirements. The program will read in an HSF model with specified system-level requirements. By accessing and processing post-simulation data, the code will determine whether or not the requirements have been met. Upon verification, the code will produce a file indicating the results of its analysis. The purpose of this code is to expand the requirement analysis capabilities of HSF beyond the subsystem level and offer HSF users insight into higher-level system operations and statuses during the entire simulation.

1.2 Objective and Deliverables

The two major objectives of this thesis were: to construct a means of connectivity from SysML to the Horizon Simulation Framework and to endow HSF with the ability to assess system-level requirements. The resulting deliverables would provide evidence that the thesis objectives were achieved. Each deliverable had a purpose related to either meeting an objective, or providing confirmation that an objective was met. A list of the deliverables is shown in Table 1 below.

Table 1: Thesis Deliverables

Deliverable	Purpose
HSF Translator Plugin	To translate SysML files into HSF-readable input files
Plugin-Generated HSF Input Model	To provide visual confirmation of HSF Translator Plugin operability
HSF Simulation Output Data	To provide confirmation of plugin conversion success via data comparison
HSF System-Level Requirement Analyzer	To analyze requirements at the system-level using post-simulation data

HSF Translator Plugin

One of the main deliverables of this thesis is the HSF plugin that deciphers specially-formatted SysML models into input files readable by the Horizon Simulation Framework. This plugin will assess a model, created and properly formatted in SysML, and build an equivalent model, specially tailored for processing in the Horizon Simulation Framework. Essentially, this plugin will grant for cross-platform support of schedule generation for systems. This cross-platform capability will allow users, specifically students, to generate visual representations of complex systems and subsequently model potential system behavior. The efficacy of the HSF translator plugin will be initially gauged by comparing the newly-created HSF file to the original HSF file upon which the SysML file was based. Further efficacy will be measured by running the plugin-generated model in a simulation and comparing the output data with that of the original simulation.

Plugin-Generated HSF Input Model File

An output file generated from the Horizon Simulation Framework Plugin will also act as one of thesis deliverables. A SysML-created model will be processed in the HSF Plugin. This model of a spacecraft is based upon an existing test case from past Horizon Simulation Framework theses. The plugin-created file will successively be compared to the original HSF input file as a method of measuring plugin effectiveness. The resultant output file will act as an initial litmus test, indicating that plugin operations were successful.

HSF Output Data

The Horizon Simulation Framework output data is an additional deliverable that will measure translator plugin competency. The plugin-translated SysML file will be used as an input for the Horizon Simulation Framework. HSF will run a simulation of the model, generating feasible schedules that function within system parameters and constraints. Outputs from the simulation will take the form of dataset files of different system parameters. Each dataset file will represent a series of parameter values indicating the state value at consecutive time steps. This deliverable will act as another method of verification that the plugin is an effective translator of SysML files. Verification would be achieved if the dataset values from the plugin simulation match the values of the original simulation.

HSF System-Level Requirement Analyzer

A system-level requirement analysis module is the final deliverable of this thesis. The module will accept defined system-level requirements and verify system adherence to the state requirements. This module will be tested by crafting sample requirements in the HSF model and running them through the analyzer module. To confirm accuracy, the requirement analysis will also be calculated manually. Requirement analysis will be deemed successful if results from the hand calculation match those from the module calculation. The assertion of a requirement success or failure would be checked as well to assess the analysis capabilities of the module.

1.3 Novelty

The ability to link SysML to HSF is a new concept which will increase the capabilities of SysML models while simultaneously adding to a list of acceptable inputs for HSF simulations. Instead of spending additional time re-conceptualizing and remodeling a system for use in HSF, users can take advantage of the plugin to reconfigure the SysML model almost instantly.

Students can exploit model-based systems engineering to better understand the design process. Especially in senior capstone design courses, model-based systems engineering can be an effective tool in helping students design intricate systems and verifying their requirements.

MBSE through this SysML/HSF connection has the capacity to help students conceptualize a system early on in the design process, and then determine whether or not it can fulfill requirements given the constraints and subsystem parameters. SysML currently isn't a part of Cal Poly's systems engineering curriculum, but linking it to a platform that has been created through the efforts of multiple Cal Poly Master's theses can open a door that will entice students to experiment with the application, expand their skill set, and develop experience with different model-based systems engineering software programs.

It is crucial for educational institutions to be on the cutting edge of industry practices and technology. University adoption of model-based systems engineering, specifically SysML, can help students gain the competitive advantage required for working as systems engineers in industry. Creating this connection to SysML through an existing Cal Poly MBSE resource, the Horizon Simulation Framework, will offer further enticement to work with this composite system modeling language.

The prevalence of model-based systems engineering has intensified with the development of increasingly advanced systems, and with it, usage of various modeling software types has become a necessity. Microsoft Visio helps users generate flowcharts and production schedules for a system. PTC CREO allows users to create a visual representation of the physical system as a series of subassemblies and components. MatLab is an excellent tool for multi-disciplinary optimization and system data processing and visualization. STK has the ability for users to conceptualize and visualize flight paths for spacecraft systems. These programs are only a few examples of the countless system modeling software suites that can portray complex systems in different ways. Each suite has its own set of advantages and disadvantages. As George E. P. Box once said, "All models are wrong, but some are useful." To increase the usefulness of these different modeling methods, there is a need for interconnectivity between them.

Facilitating connectivity between models makes it easier for systems engineers to simultaneously visualize different aspects of the same system. Each system modeling program

can only offer a limited perspective of the system. However, by creating a link between models, the capacity for helping design improved systems increases dramatically. The ultimate goal is to have a single resource that possesses the capabilities of all modeling software and model virtually any aspect of a system. This centralization of MBSE would prove useful all systems engineers, acting as a "one-stop shop" for system modeling. Though we have yet to see this come to fruition, linking current modeling software at lower levels is the first among many steps towards accomplishing this goal.

Historically, HSF has solely analyzed constraints at the subsystem level. During simulation and schedule generation, HSF checks calculated state variable values at each time step to ensure they comply with the user-defined constraints. In an HSF simulation, all subsystems are independent of one another. The system-level requirements analyzer offers users a means of assessing system performance that requires investigation into multiple subsystems. The requirement analyzer can also evaluate system activity over the course of the entire simulation. Instead of examining system properties at a single instant, the module can observe how the system behaves throughout the span of the simulation. This is crucial as effective systems engineering relies on studying an entire system and how its parts work together, not just the individual actions of its components or properties at particular time steps. Ultimately, this ability to verify at a larger scale can help systems engineers determine how well a system can comply with system-level requirements, making the Horizon Simulation Framework an even more powerful tool.

1.4 Summary of Sections (Thesis Overview)

This thesis will discuss the thought process, design, and creation of the Horizon Simulation Framework Translator Plugin and the system-level requirement analyzer. A foundation of Model Based Systems Engineering leading into SysML and the Horizon Simulation Framework will first be established to provide inside into the development of the plugin. This thesis will then discuss the deliverable codes and the involved decision-making processes,

including justification for the plugin's and module's respective algorithm logic flows. The tools and libraries utilized in code formulation will also be investigated. This thesis will also review the requirement verification processes, test cases, and problems encountered for both the plugin and the module. Next, the results of the output data will be visualized and discussed. Finally, this paper will analyze the results, generate conclusions, and propose some potential future work for the Horizon Simulation Framework, the translator plugin, and the requirement analyzer module.

BACKGROUND AND RELATED WORK

2.1 Model Based Systems Engineering

In order to better understand model-based systems engineering, it is crucial to first have a grasp of systems engineering. Systems engineering is a design process which many areas of expertise are utilized to develop an efficient solution that fulfills the needs of different stakeholders to resolve a complex problem. The process necessitates the synergy of knowledge between technical and managerial processes in order to formulate a solution that can handle complexity but retain balance among its subsystems. [3]

In systems engineering, knowledge of management skills and processes are needed to oversee that system development is on schedule, within budget, and meets the customer's needs. Tasks relating to these management responsibilities include, but are not limited to allocating technical resources, mitigating risk, and overseeing system performance and requirements. The role of technical skills and processes within systems engineering is to establish system requirements and performance parameters, design the system, and verify that the system meets the established requirements. This is just a simplification of the roles and responsibilities of systems engineering and in no way is a limiting definition. As the demand for system capability and performance increases, so must systems engineering improve and evolve [3]. With model-based systems engineering, model use and manipulation is a responsibility which falls under managerial and technical roles alike.

Model-Based Systems Engineering, or MBSE, is the use of models and/or modeling resources to facilitate systems engineering tasks. Such tasks include verifying requirements, validating system specifications, and analyzing system performance. [3] These are only a few examples of MBSE application as the practice can be applied for the entire duration of the system's life cycle. [3]

As opposed to document-based systems engineering, MBSE enlists the aid of models to delineate the systems development process. The practice combines different modeling fields and

aspects of the system throughout the systems life cycle. MBSE covers development at all hierarchical levels, from a system of systems (SoS) all the way down to the component level. [9]

To effectively grasp the concepts behind Model-Based Systems Engineering, the role of the model must be understood. A model is the manifestation of a system that that allows the user to visualize or interpret the system with respect to at least one of its aspects. Models can be used to exhibit at least one specific concept of the system, but are not meant to show every single feature within the system. [3] There are no limits to the types of models and the characteristics they can communicate to users as well as the purposes they serve.

Models are extremely versatile in their uses. System models give users a medium to effectively record and organize subsystems and components. They can also be used for defining systems of all types, be it a completely new system, a previously established system, or even a system modified from a pre-existing one. The utility of models is constant throughout the entire system development cycle. Early on in the design process, they can act as retainers for validating and specifying requirements. They can also act at preliminary representations of conceptual designs. Models can be compiled to encompass multiple system aspects, especially when subsystem relationships and their requirements generate a need for interconnectivity. Models can also allow for engineers to trace the flow of requirements from component-level, all the way up to system-level as well as a top-to-bottom approach. [3]

In addition to providing a visual representation, models can also provide a means of evaluation. System development teams can utilize combined models to conduct high fidelity simulations and analyses. [3] Engineers can utilize models to perform tradeoffs between various system designs. System attributes and performance requirements can be analyzed through the use of models. Engineers can also take advantage of models in the verification process to ensure the system fulfills specified requirements and meets stakeholder needs. Even in the case of changes to the design or requirements, models can act as a visual representation of the affect the changes have on the system. [3]

MBSE is an evolutionary change in the systems engineering process. Instead of design specifications and requirements being maintained through documentation, maintenance is conducted through the development and use of a working, cohesive model that effectively represents the system. Engineers can gain numerous advantages from MBSE implementation. Using models can allow for system and requirement complexity management while effectively disseminating information among the team and stakeholders. Simultaneously, models can reduce the time needed for system development and increase the quality of the system design. Duties that have historically been conducted through documentation have become much easier through the use of MBSE. Consequently, model-based solutions to systems engineering are rapidly overtaking the traditional process of documentation tracking. This is hypercritical, especially in this swiftly-changing engineering landscape of progressively intricate systems. [3]

2.2 Horizon Simulation Framework

The Horizon Simulation Framework, or HSF, was the result of a Cal Poly Master's thesis dealing with model-based systems engineering. HSF is a code base used for system simulation and subsystem constraint verification. With the framework, a user can model a system, an amalgamation of its subsystems, parameters, initial conditions, constraints, and dependencies along with inputs for environment and simulation parameters. Once executed, HSF handles these inputs to conduct an analysis, searching for working schedules that satisfy constraints given the quantified parameters and initial conditions. These schedules are a compilation of an initial system state and events that follow it, as generated in a simulation, showing system performance over a specified period of time. HSF ensures that each state must fall within the specified system constraints.

A series of potential working schedules for the specified system is HSF's main output. The framework performs the simulation by employing an exhaustive search algorithm and cropping the amount of schedules per the user's direction. HSF can generate multiple schedules if the user defines it in the simulation parameters. Each schedule is an original "day-in-the-life"

simulation of the system depicted via state data. One can track the values of defined parameters, or state variables, over the course of the simulation. The Horizon Simulation Framework is the critical source that connects constraint verification, system modeling, and scheduling all in one location. [8] Once executed, HSF will comprehensively search for a solution, until the number of working schedules is reached, or until a preset amount of time. Upon completion of the simulation, HSF will provide data for a working solution; otherwise the program will notify the user that no working solutions were found. This found solution is not the optimal solution, only a working one. [8]

The main scheduling algorithm and the system model are the two main components of HSF. These two parts are independent of each other so that their interaction does not influence the effectiveness of the simulation. For example, if desired, the current Main Scheduling Algorithm (exhaustive search) could be replaced with a different scheduling algorithm. The effectiveness of the new scheduling algorithm could then be tested with the system model being the constant. The other main component, the system model, is made up of smaller subsystem models, all of which are independent of one another, maintaining another level of modularity. Modularity enables the user to swap out current subsystems for ones with different parameters or operational functions. The figure below visually depicts this modularity and interaction between the various HSF components. [8]

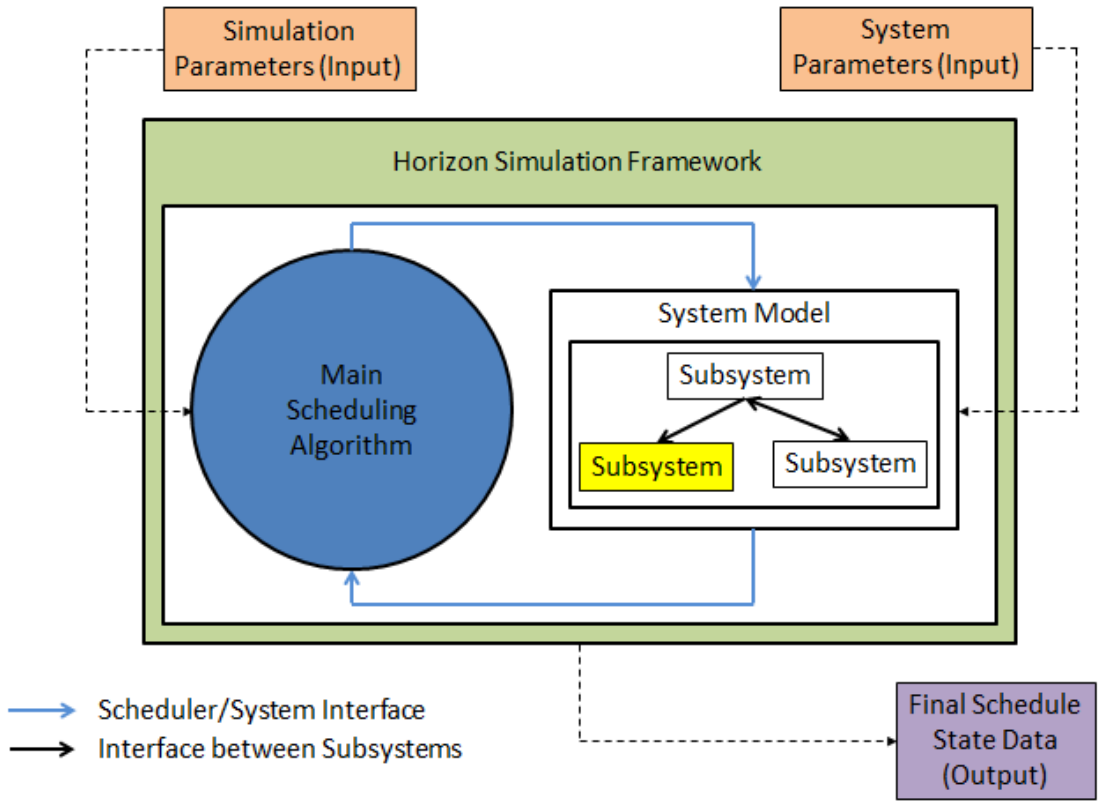

Figure 1: Major Modules and Interfaces Implemented in HSF [8]

When generating a schedule, HSF must decide whether a task is acceptable for the schedule by submitting it to three tests. The series of tests essentially asks the following questions:

1. Has the first task been completed by the system?
2. Can the same task be repeated?
3. Given the current state and parameter values, can the system operate within the specified subsystem constrains and execute the task?

If the task successfully passes the tests, the schedule adds the new task and begins the task-testing procedure anew. [8]

One part of HSF's functionality is ensuring that the modeled system satisfies user-defined constraints, which are limitations that have been enforced on subsystems and state values. In

order for a schedule to be saved, all constraints must be satisfied at every step, and there must be viable tasks until the simulation end time. Currently, HSF uses the constraint checking cascade as its method of choice in deciding if a defined system is capable of performing a task or action that the system executes during a time step in the simulation. This algorithm first amasses the subsystems that generate data for the constraint in question. Each subsystem is executed, adding the relevant state data in the process. This state data is the means of data storage for the simulation, acting as a vector that stores all system parameter information for the series of time steps that occur during the simulation. The constraint qualifier is then evaluated after the execution of all relevant subsystems. Upon meeting qualifications, the constraint passes and starts again with the next constraint. The figure below shows the process HSF undergoes when checking a constraint. This continues until all of the system constraints have been checked. Upon completion of all constraint checks, the scheduler executes the remaining unconstrained subsystems. Passing all defined constraints permits the addition of the task and state variable data to the schedule, making it an event, the simplest unit in a schedule. The next potential task is then analyzed for possible addition to the schedule. Figure 3 shows a diagram of the constraint checking cascade process. [8]

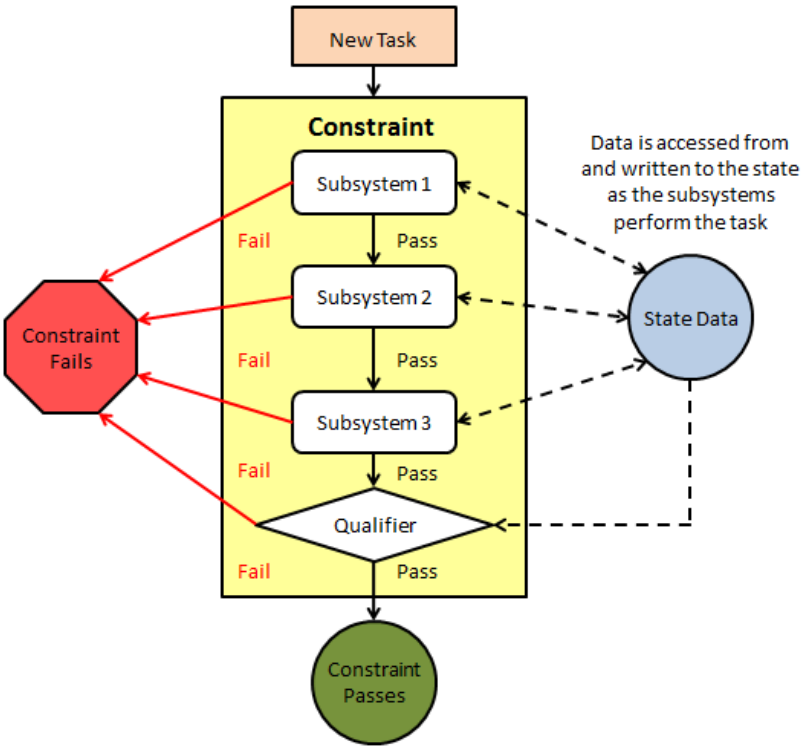

Figure 2: Internal Constraint Functionality [8]

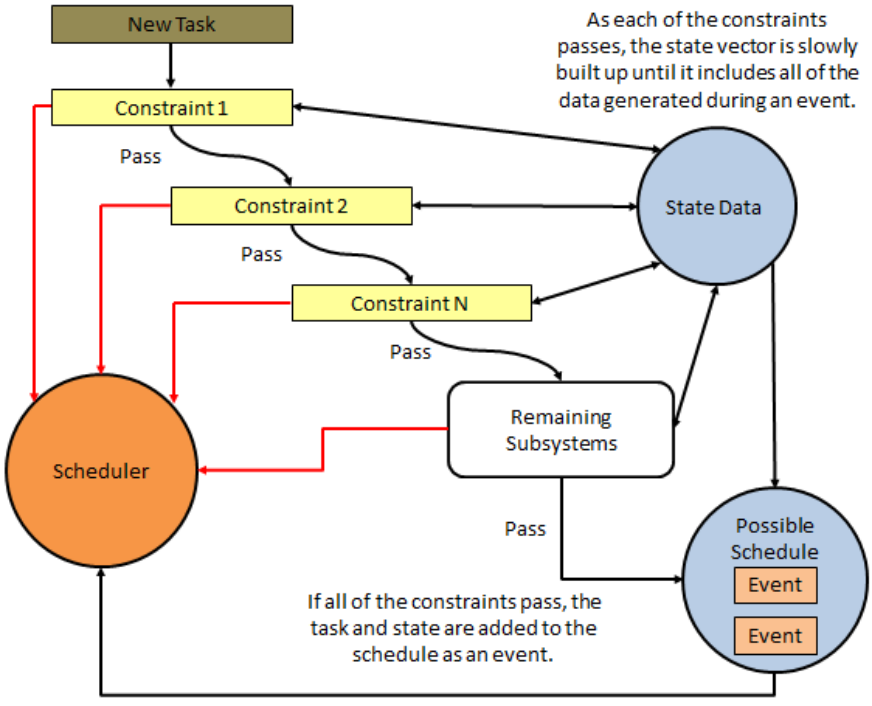

Figure 3: Constraint Checking Cascade Diagram [8]

The concept of operations, or ConOps, is the statement of a system's task or tasks as defined from the stakeholder. Essentially, the ConOps specifies what the system is supposed to do, but does not describe how it should complete its objective(s). [3] The intention for HSF's development was to allow design teams to learn about verification of system requirements and iteration in the early stages of the design process. Subsequently, it was also meant to show system engineers the importance of the system environment, subsystems, and ConOps development during these design sub-processes. [8]

The function of the Horizon Simulation Framework does not involve Multi-Disciplinary Design or Optimization, or MDO. MDO, like its name suggests, tries to find the best and the most efficient design. The process places an emphasis on a specific subsystem parameter or parameters and tries to minimize said parameter. Contrary to MDO, HSF's goal is to analyze a given system, determine if the system is feasible, and verify that the system satisfies the stated requirements. [8]

The Horizon Simulation Framework takes a modeled system with subsystem parameters, initial conditions, and constraints, and tries to find a working solution within the giving environment. There is neither a process nor objective for finding an optimized solution. The Horizon Simulation Framework simply wants to find a working solution, verifying that the modeled system can indeed operate within the stated constraints given its parameter values and initial conditions.

One outstanding quality of HSF is its modular interface, which enables users to swap out subsystems, which are one of the simplest elements a user can model. Subsystems can be modeled such that they are independent of any specific simulation or subsystem. A user can code up a subsystem with its own functions and parameters and drop the new subsystem into the framework. From this, an impact analysis can be conducted using state data on how the new subsystem affects system performance.

As previously stated, the goal of HSF is to find a working solution, not an optimal one. That being said, the fidelity of the data is highly dependent upon the person or persons developing the model. When it comes to the usefulness of its output data, HSF exhibits a "garbage-in, garbage-out" philosophy. The user receives the same value of data in which they put into the simulation. Creating an accurate model of the system will reward users with beneficial and useful data, while a poorly modeled system will spew forth information that is not as valuable. This is why it is crucial for users to understand HSF's internal operations so they create effective system models that provide useful simulation data. [8]

One benefit of the Horizon Simulation Framework is its flexibility to operate with any system. This flexibility lies in the framework having no dependency on knowing the background behind the modeled systems and subsystems. Utilizing the previously-modeled subsystems as a resource, users can quickly simulate the system without having to build it up from scratch. They can also analyze the system quickly without having to sacrifice the fidelity of the data. [8]

In order for HSF to operate properly, the system must adhere to a series of strict conditions. First, the system cannot possess any circular dependencies. Dependencies are the interaction that subsystems have between one another. Acting as filters, dependencies dictate which data can be flowed down to the next subsystem. In HSF, all dependencies must flow in a linear fashion in which each subsystem relies on the data on its predecessor. This leads into the second condition: verification of subsystems must occur in the proper order. This order is the same as the dependencies. The final condition is that during the simulation, generated state data must be accessible to all subsystems. [8]

1. HSF Inputs

The Horizon Simulation Framework receives three major input files for a simulation, all of which are in XML format. The first file establishes the constraints of the simulation's operation, all contained within an overarching scenario definition, as seen in the figure below. This object is then broken down into two child elements to define operational parameters for the

simulation and the scheduler. The “simulation_parameters” tag states the simulation start time and end time as well as the starting Julian date. The “scheduler_parameters” tag” sets the time step, or increment by which the simulation is propagated. The last piece of information included in this file is maximum amount of schedules. This value is the cap at which HSF will crop any extra schedules.

```
<?xml version="1.0"?>
- <SCENARIO scenarioName="testScenario">
    <SIMULATION_PARAMETERS SimEnd="500.0" SimStart="0.0" SimStartJD="2454680.0">
      </SIMULATION_PARAMETERS>
    <SCHEDULER_PARAMETERS SimStep="30.0" numCrop="2" MaxSchedules="2">
      </SCHEDULER_PARAMETERS>
</SCENARIO>
```

Figure 4: HSF Input for Scenario Parameters

The second input file for HSF is the targets file, which is a database for all of the desired targets in the simulation. For this case, any given target could be either one of two types: a ground station or an image. An example of both target types can be seen in the image below. Each target has a set of parameters, crucial in the scheduler’s system analysis. These parameters are described in the table below.

```
- <TARGET Value="-1" MaxTimes="1000" TaskType="CommTask"
  TargetType="FacilityTarget" TargetName="groundstation4">
    <POSITION ICs="Matrix(3,1,{-29.15, 114.56, 0})" PositionType="StaticLLA">
      </POSITION>
</TARGET>
- <TARGET Value="3" MaxTimes="1000" TaskType="ImagingTask"
  TargetType="LocationTarget" TargetName="imagetarget1">
    <POSITION ICs="Matrix(3,1,{39.2945, -71.1559, 0})"
      PositionType="StaticLLA"> </POSITION>
</TARGET>
```

Figure 5: HSF Input for Target Examples

Table 2: Target Parameters and Descriptions

Target Parameter	Description
Value	Target priority number designation. The lower the integer value, the higher the priority. -1 is the highest priority target.
MaxTimes	The duration of time during which the system can interact with the target.
TaskType	Indicates what kind of interaction task the system has with the target (imaging or communications).
TargetType	Indicates the type of target (imaging target or ground station).
TargetName	Reference designator of target
ICs	Target location described via 3x1 matrix-dimensional coordinates
PositionType	Reference coordinate system

The last input file is about the system itself. This file states the system location, subsystem parameters, initial conditions, dependencies, and constraints. HSF uses all three of these files to execute a simulation and produce a set of schedules, all of which converge to working solutions given all of the system parameters and obey constraints. This final file will be discussed later as it is the primary focus of this thesis' deliverables.

3.1 **Aeolus Test Case**

Named after the Greek god of wind, Aeolus is a weather imaging satellite system. [8]. The system's objective is to take pictures of specified targets while in orbit, and relay the data back down to ground stations. Aeolus has a predetermined list of targets. Each target has a unique location and priority. The satellite gives preference to targets with higher priority during data acquisition. Aeolus then stores the information and then sends it back down to a ground station at the appropriate time. Like the image targets, ground stations have been previously specified and stored prior to the start of the simulation. The figure below depicts the mission concept highlighting the ground path and targets for the Aeolus mission.

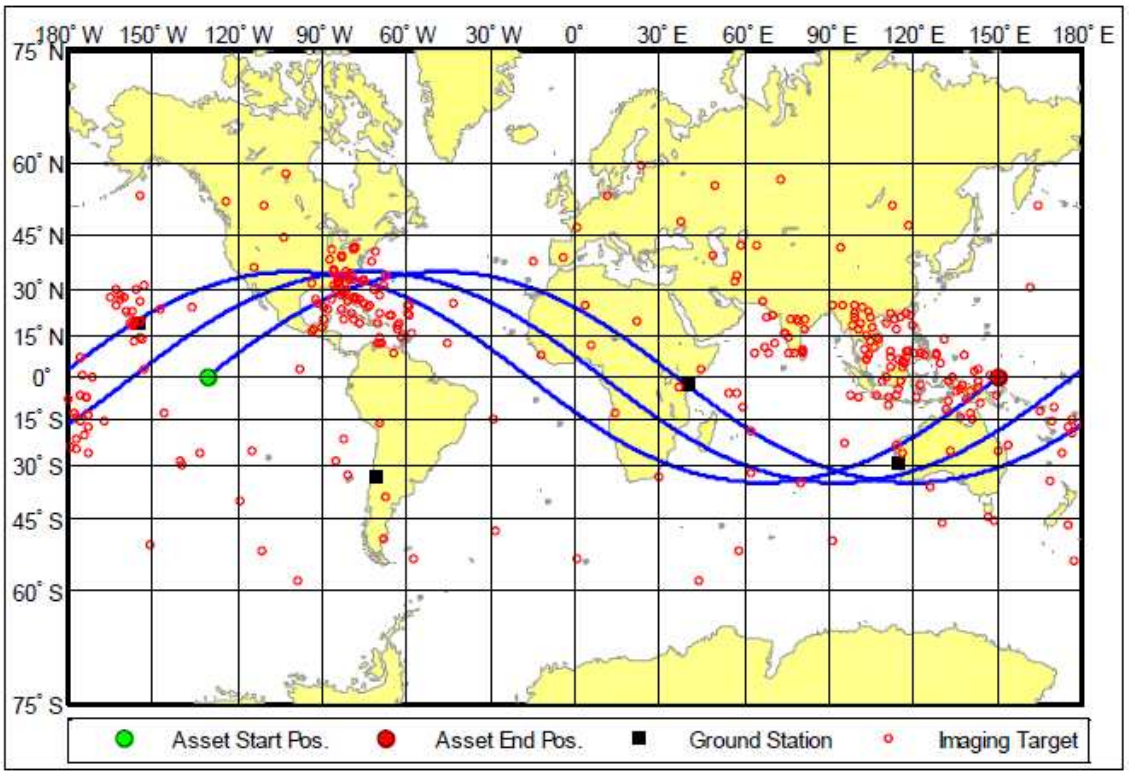

Figure 6: Aeolus Mission Concept [8]

The Aeolus system had already been modeled for the Horizon Simulation Framework. This fictional system has acted as a baseline test case for past Horizon Simulation Framework theses. This previously established relationship with HSF would make it easier to verify translator plugin operability. The Aeolus system is defined by six subsystems, each of which has its own subsystem parameters, constraints, dependency functions, and state variables.

Aeolus was remodeled in SysML as the product of another thesis project, *SysML Based Cubesat Model Design and Integration with the Horizon Simulation Framework*. The SysML model contained all of the subsystem information as the original model. Exocube was also modeled as a result of the thesis, but verification of successful plugin conversion could not be established and analyzed unless there were an existing HSF input file and series of previously modeled Exocube subsystems within HSF codebase.

Successfully converting the Aeolus SysML file would act as one of the indicators that validated proper operation of the plugin. The other portion of plugin operability validation would involve running a simulation of the newly converted Aeolus model, and comparing it to the simulation data of the original HSF Aeolus simulation. Because a precedent has already been established with the Aeolus system, a match in the simulation data would further confirm that the conversion was successful.

The HSF inputs needed for proper translation and simulation were contained in the BDD, or block definition diagram. This diagram modeled three aspects of the mission: the spacecraft system, the ground system, and the environmental system. The spacecraft system defined Aeolus, the system being investigated. The ground system defines the targets on the Earth's surface. The environmental system defines the external conditions which act upon the spacecraft system.

The environmental system block contains conditions, such as the initial spacecraft position and velocity vectors. The spacecraft system block is broken down further into six blocks, each representing one of Aeolus' six subsystems. Each sub-block describes various subsystem parameters, including subsystem ID's, initial conditions, constraints, and dependency functions.

An IBD, or internal block diagram was created of the Aeolus spacecraft system, which housed the six defined subsystems. Each of these defined subsystems had its own BDD, or block definition diagram. Each BDD describes various subsystem parameters, including subsystem ID's, initial conditions, constraints, and dependency functions. Some BDD's were broken down even further, separating the subsystem into components.

2.3 SysML

In order to understand the inner workings of the HSF Translator Plugin, some background about SysML must be established. SysML, or Systems Modeling language, is an outlet of MBSE that can be used to represent and develop systems at various stages of the system development process. [3] SysML finds its origins in the second version of UML, or Unified Modeling Language, which is the usual the standard of modeling language for software. UML

was selected for SysML support because of its robustness and the fact that it fulfilled many of the needs of the systems engineering community. [3] UML 2 was also supported by a wide range of industry tools. SysML is utilized in helping design and specify systems. It can also be used in calling out hardware and software components which can be flowed down to be designed in VHDL and UML respectively. [3] SysML lets users to define systems of all types. SysML allows for models a variety of systems including, but not limited to software, hardware, facilities and procedures. It's applications in industry knows no bounds. [3]

Incorporating SysML into the system design process brings forth many capabilities. A variety of items within the system can be represented. SysML can be used to model a variety of system entities. Examples of such entities include models of system structure, requirements, constraints, and behavior. The possibilities are endless when it comes to the system modeling capabilities of SysML. What makes SysML a powerful tool is the ability to model relationships between all of these system aspects through the use of diagrams. [3]

A SysML diagram is a visual depiction of a specific aspect of the modeled system. It's a representation of a smaller portion of the entire system. [3] In all, there are nine different types of SysML diagrams, with each type dictating the kind of items and other model elements that are allowed in the diagram. [3] Diagram taxonomy and breakdown are shown in the figure below. A SysML diagram has three major components that represent the system's behavior, requirements, and structure respectively. A Requirements Diagram, as its name states, shows the system's requirements and how they relate other system entities. The purpose for this diagram is to allow users to visualize and recognize how requirements were originally flowed down. The behavior component is further broken up into four diagrams: Activity, Sequence, State Machine, and Use Case. The structure component is broken down into four diagram types as well: Block Definition, Internal Block, Parametric, and Package. [3] The functions and purposes of these diagram types are discussed in further detail in *SysML Based Cubesat Model Design and Integration with the Horizon Simulation Framework*.

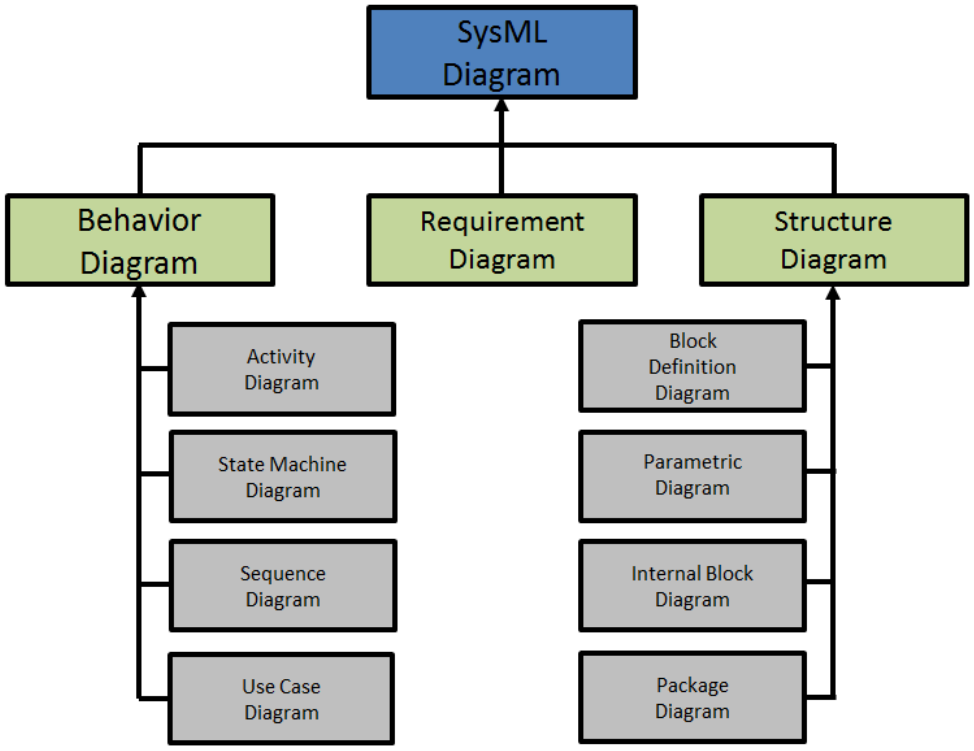

Figure 7: SysML Diagram Taxonomy [3]

HSF and SysML are both mediums through which users can implement MBSE methods, but they possess their own unique sets of strengths and weaknesses. Their outputs vary as do their uses. MBSE is useful in visualizing relationships between subsystems, components, parameters, constraints, and requirements. HSF is ideal for simulating system performance and verifying satisfaction of system requirements. Knowing when to use either tool is critical in MBSE, making it even more useful to create a means of connection between the two. A breakdown of differences between SysML and HSF is shown in the table below.

Table 3: Breakdown Between SysML and HSF

SysML	HSF
Visually represents a model and their relations	Capable of providing system data to be graphically represented
Output is a series block diagrams	Output is a series of data
Generates a static representation of a system	Generates a series of schedules that represent system state and operation for a specified period of time
Model allows for free flow of system representation	Model is restricted by cascade structure of subsystem dependencies, constraints, parameters, and state variables.
Stores requirements, parameters, constraints	Verifies whether or not system can meet stated requirements given system parameters.

2.4 Literature Review

2.4.1 Applying SysML MBSE to an HSF Model

SysML Based Cubesat Model Design and Integration with the Horizon Simulation Framework is a thesis focused on modeling systems in SysML. The intent was to create a SysML model format that could be transcribed into a layout which could be interpreted by the Horizon Simulation Framework. The thesis organized an investigation a system modelling that could ultimately communicate with HSF while simultaneously offering the advantages of a SysML model. [5] Modeled systems included Aeolus, the Horizon Simulation Framework test case, and ExoCube, a real-world CubeSat program. The Aeolus model will be discussed later on in this thesis, as it was the basis for verifying that the HSF plugin could successfully adapt a SysML model for use in an HSF simulation. The rationalization for modeling the ExoCube mission was to demonstrate the application of SysML modeling methods to a tangible system. [5]

The study made several conjectures in its approach. One such assumption was that the SysML model was organized in a consistent fashion. All representations of a particular item type within the system were expected to be expressed in the way. Any items that failed to obey the established formula risked omission from the translated result. Another assumption made prior to analysis was that the HSF Translator Plugin would operate as anticipated. Like any program,

overlooked bugs within the code could jeopardize the intended operations, potentially omitting system components, incorporating unwanted parameters, or even failing to create a file altogether. However, it was the continuous checking of file formatting and debugging that aided in minimizing the risks of these assumptions.

The report examined SysML and took a thorough look at the steps taken in constructing the SysML models. MagicDraw was the SysML software client used in modeling both missions. [5] The "6 C's of Model Based Systems Engineering" were utilized as a modeling rubric. Each "C" represents a quality that ensures that the model is serving its purpose: Changeability, Completeness, Comprehensibility, Confinement, Consistency, and Completeness [5]. The study then gave overviews of each SysML model and the logic involved behind their structure and the relationships between the system and its features.

The Aeolus SysML model was tested using the deliverable from this thesis. The HSF Translator Plugin would refine the model into a scheme suitable for the Horizon Simulation Framework to understand and simulate. [5] The study took on a two-pronged approach for authenticating the model's success, both of which will be examined further in the Results Section of this thesis; visual comparison of the original and transcribed models, and numerical comparison of both model's simulation data.

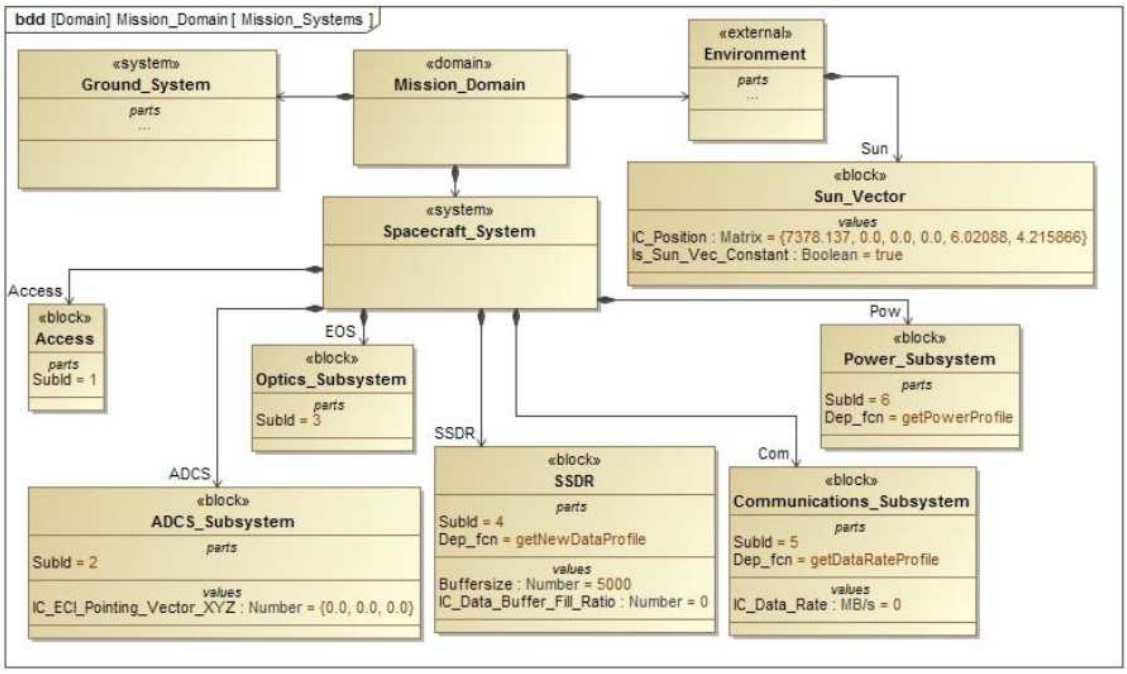

Figure 8: Initial Aeolus System SysML Block Diagram [5]

The results of the thesis concluded that the applied SysML modeling technique was compatible with the HSF Translator Plugin. Loading the SysML Aeolus model into the plugin yielded a successful transcription. When juxtaposed to the initial HSF model, the converted model produced a configuration which perfectly mimicked the original design. [5] Data presented from the subsequent simulation moreover affirmed validity of the translated model. [5]

The study shaped the ideas that eventually became the building blocks for this thesis. It posed a question of whether or not a system represented in SysML could be assembled specifically with the intention of simulation in an alternate MBSE program. The paper explored modeling to ultimately translate, while this thesis grabbed the baton with the goal of conceiving the means of translation.

2.4.2 Verification of Requirements for a SysML Model

Similar to this thesis, *A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints* is a conference paper that investigates modeling a system in SysML and adopts a secondary language

profile to simulate system operations [10]. The goal of this study was confirm successful validation of requirements by modeling an embedded real time system into a SysML diagram, and converting it into an ideal model for simulation and analysis. A computer subsystem that has a designated operation inside a much larger system is called an embedded system [11]. When an embedded system is assigned timing constraints, it becomes an embedded real-time system, or ERTS [10]. In this research paper, the ERTS example of focus is a pulse-oximeter, a tool that externally gauges blood oxygen concentration levels. Because of the device's use in surgeries, maintaining precise measurements of energy consumption and execution timing of the pulse-oximeter are extremely crucial. The researchers used the Activity Diagram functionality within SysML to define the possible operations of the system. [10]

In conjunction with SysML, a UML profile named MARTE is employed in this study. MARTE stands for Modeling and Analysis of Real-Time and Embedded systems. [10] Its purpose is to represent ERTS as a means of modeling and providing analysis support. Ultimately, MARTE has the ability to assign system specifications and parameters to system activities within the SysML Activity Diagram, allowing the model to be simulated and analyzed [10].

Time Petri Net with Energy Constraints, or ETPN, was the means by which the system was evaluated. This tool can simulate behavior of the ERTS to provide data on the pulse-oximeter's execution time and energy consumption. [10] During ETPN model evaluation, the optimal paths for best and worst constraint values are formulated. [10]

The process behind this study began with the delivery of the ERTS description, which is in turn used to construct the SysML Activity Diagram [10]. Then, MARTE was used to designate the execution time and energy consumption information to activity and activity transitions within the SysML Activity Diagram. Once the information had been designated to the appropriate locations within the diagram, the SysML Activity Diagram was translated into an ETPN model [10]. After translation, simulation of the ETPN model commenced, providing schedule data as well as state data of the constrained values. These resulting values were then compared against

the initial requirement values from the beginning of the process. A successful compliance with system requirements ends the process. If the simulation data failed to adhere to the specified requirement values, system parameters were adjusted and the process started again from the beginning. [10] This process was repeated until a working solution is realized. A block diagram of the process can be viewed below.

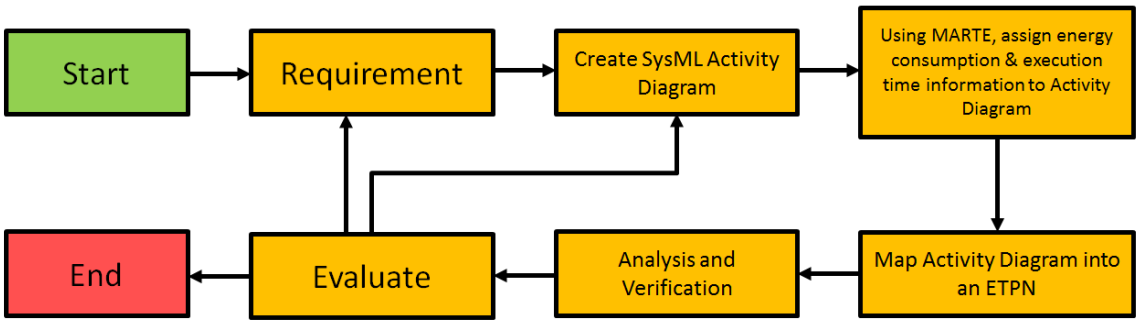

Figure 9: Process Used in Evaluating Requirements in SysML Models Mapped to ETPN [10]

The researchers made the assumption that all activities, transitions, and corresponding properties within the SysML Activity diagram were successfully translated into the ETPN model. There is a possibility that any activity could have been improperly translated or even omitted from the ETPN model, and ultimately ignored during model evaluation. The same assumption was applied to MARTE in its ability to assign the system specifications to the appropriate activities and transitions.

After translation and evaluation of the model, the simulated values were compared to actual measured hardware values and constraint values. Comparison indicated similarity between the values and adherence to specified constraints for both energy consumption and execution time [10]. The results of this study showed the successful translation of the SysML Activity Diagram into a working ETPN model. Successful application was proven from evaluating the ETPN model, showing that requirements could be evaluated. Research will eventually extend into translating other diagram types within SysML.

Parallels between ETPN model and HSF can also be drawn, as they both are used to simulate system behavior in the form of schedules. One major difference, however, is that HSF provides simulation data of any working scenario, as long as the constraints are obeyed. ETPN, contrarily, actively searches for behavior paths with the best and worst performances with regards to the specified constraints. [10]

Another similarity between this thesis and the research paper is deduced through the testing processes. Both involve initial modeling in SysML, translation to another model format for system simulation, and evaluation of system requirements. One difference however, is that the research paper makes no indication about means of translation. In its conclusion, the paper hints at dedicating future research into automating the SysML-to-ETPN model creation process, implying that original translation was conducted manually. Another noted difference was that the paper's references to requirements posed upon the execution time and energy consumption parameters were akin to subsystem-level constraints in HSF. There was no investigation of system-level requirements, which rely on system performance throughout the simulation period, not just a single instant.

2.5 Problem Restatement

The *SysML Based Cubesat Model Design and Integration with the Horizon Simulation Framework* thesis demonstrated a new means of model representation, but the block diagram representation of the system was the extent of its capabilities. A means of analyzing the system was still needed. The SysML model could display subsystems and their parameters but it did not give enough insight to the user on an operations level. It lacked the ability to simulate system performance. The Horizon Simulation Framework was needed to give the user a better idea of system behavior. In order to do this, a link needed to be made.

Another similar area in which SysML lacks is system-level requirement verification. Because of its inability to simulate, there is no way to mathematically confirm that a system adheres to specified requirements. HSF is also currently weak in this aspect since its focus is on

state data and ensuring subsystem constraints are met. This leads to the question: “How can a model-based engineering tool be used to assess system-level requirements?”

DELIVERABLES

3.1 Horizon Simulation Framework Translator Plugin

3.1.1 Requirements and Constraints

The objectives of this plugin were to read in a SysML file that represents a system, locate and fetch the necessary parameters and their respective values, and reformat the data into an arrangement readable by the Horizon Simulation Framework. Though it seemed as if the code had a simple objective, the means of accomplishing this objective would require complex work and logic.

One established requirement was that the code would only deal with the system model aspect. The plugin would not process any inputs pertaining to target data or simulation parameters. This would only make it necessary for users to model the system in SysML and adjust simulation and target parameters separately. Leaving the target and simulation parameters alone also created a control for testing the original and converted cases in which both systems could be simulated under the exact same conditions.

Originally, the provided SysML file was generated as a result of *SysML Based Cubesat Model Design and Integration with the Horizon Simulation Framework*. The system model was represented in the form of a block definition diagram. While this was a well-constructed SysML model of the Aeolus mission, evolution of the Horizon Simulation Framework's inner workings dictated that the model needed additional development.

One update to HSF was the addition of Python Scripting. This tool enabled HSF to encode subsystems using the Python programming language while running the simulation. HSF input files exploit this instrument as a callout and parameter. HSF had been updated to recognize this callout and execute simulations accordingly. Within subsystem dependency declarations, parameters were also established to direct HSF as to which Python function files were necessary to access during simulation.

Another update requiring the SysML model to be updated resulted from this thesis was the addition of system-level requirements. Because of the implementation of the system-level requirements analysis tool, another callout was added to the SysML model. The new node defined system level requirements that instructed HSF to process post-simulation data and indicate whether or not the system adhered to the specified requirements. With the addition of these two new nodes and parameters, and the change formatting of the HSF model flow, the SysML model necessitated revision. A zoomed out view of the revised block diagram can be seen below.

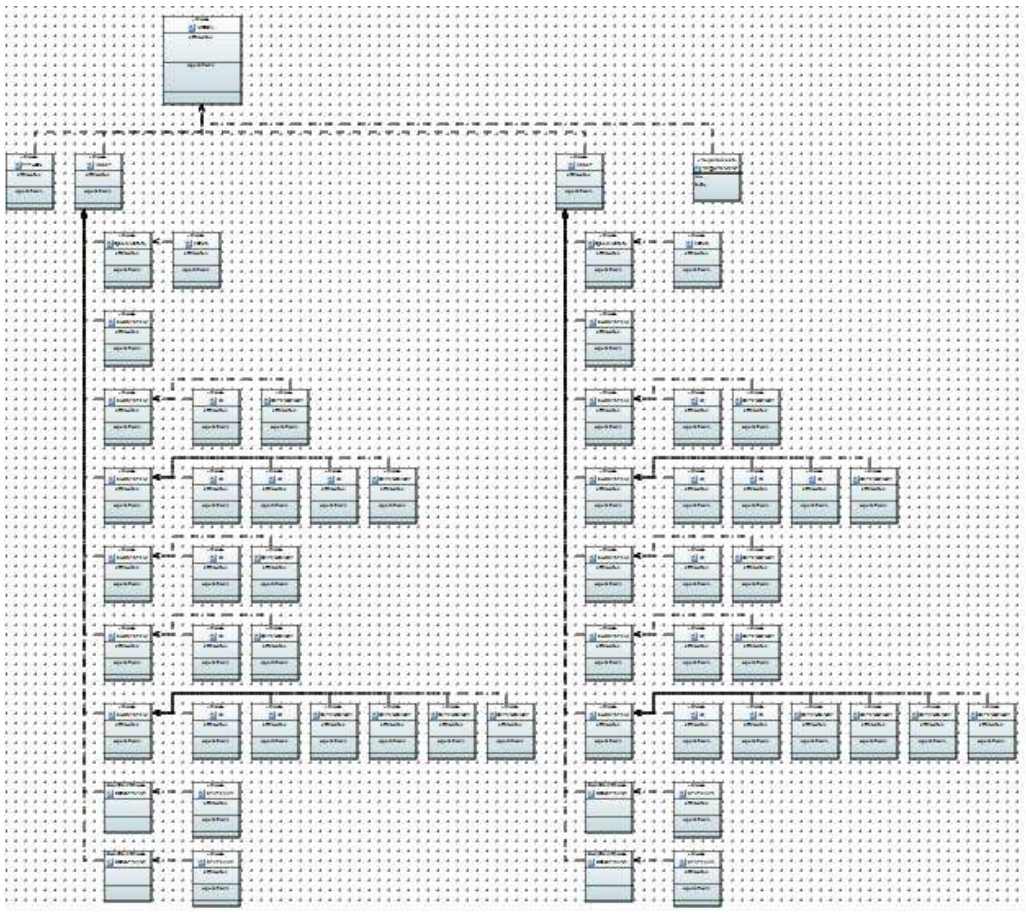

Figure 10: Revised Aeolus System SysML Block Diagram (Zoomed Out)

This revised block diagram depicts both assets as children of the model as well as the recently added Python and requirement child nodes. The updated model also exhibits each subsystems child nodes, which possess their respective parameter declarations, as shown in the images below.

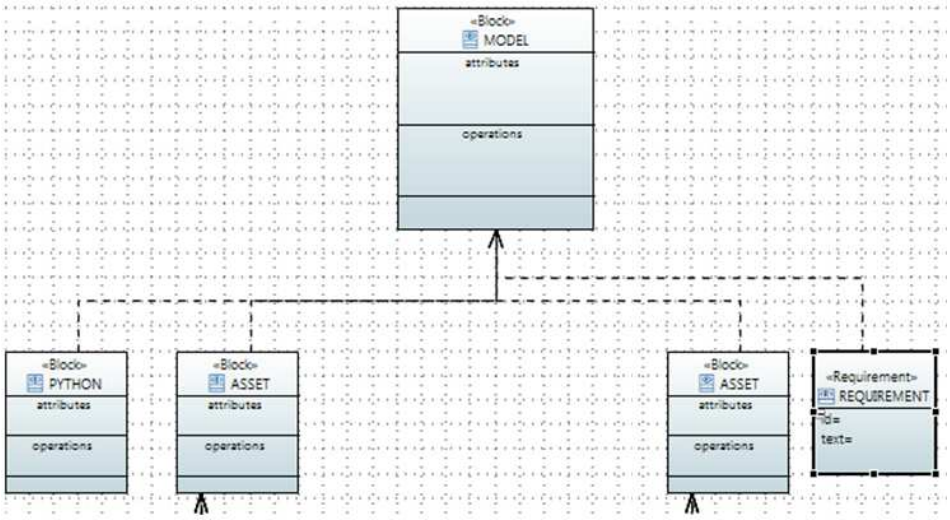

Figure 11: Aeolus SysML Block Diagram (Model Children Only)

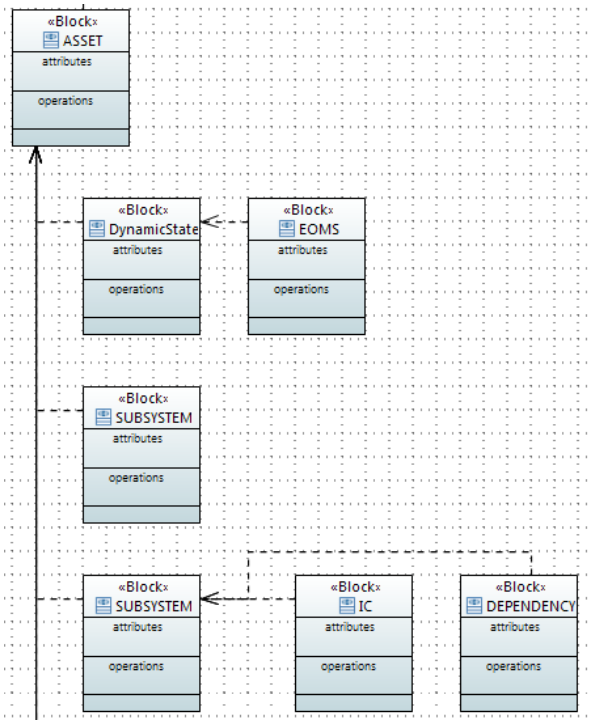

Figure 12: Section of Aeolus SysML Block Diagram (Asset Children Nodes and Subnodes)

Outside the confines of SysML viewing and modeling software, the file can manifest in the form of an XML file. The input file for the HSF Plugin is the output file generated from SysML when viewed through a web browser or any text editing and/or file viewing software. The SysML file begins with a header, stating file properties, and then proceeds to describe the

modeled system. Because of the hierarchical nature of XML, the file depicts a clear distinction between mission domains. Subsequently, there is a clear depiction between subsystems as well as their respective parameters and parameter values. Within all of the XML tags, are xmi parameters. The “xmi::id” parameter values all have extensive character strings, containing seemingly random combinations of letters, numbers, and other symbols. The other xmi parameters provide a form of classification, showing an item’s “type”, “href”, “visibility”, or “client” values. These xmi parameters were ignored as they were only pertinent to SysML. Shown below is a sample excerpt from the Aeolus SysML file, as read by a web browser. The full file contains hundreds of lines of text.

```
<uml:Model name="HSF_Aeolus_Model" xmi:id="_wtyu0PsfEeeyDfxL4E7yBg">
  - <packageImport xmi:id="_xc0YAPsfEeeyDfxL4E7yBg" xmi:type="uml:PackageImport">
      <importedPackage xmi:type="uml:Model"
        href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_0"/>
    </packageImport>
  - <packageImport xmi:id="_xd3g4PsfEeeyDfxL4E7yBg" xmi:type="uml:PackageImport">
      <importedPackage xmi:type="uml:Package"
        href="pathmap://SysML14_LIBRARIES/SysML-Standard-
        Library.uml#SysML.package_packagedElement_Libraries"/>
    </packageImport>
  - <packagedElement name="AEOLUS" xmi:id="_flJEcPsiEeeyDfxL4E7yBg"
    xmi:type="uml:Package">
      - <packagedElement name="MODEL" xmi:id="_y8hWkPsnEeeyDfxL4E7yBg"
        xmi:type="uml:Class">
          - <nestedClassifier name="PYTHON" xmi:id="_MmqqcPsxEeeyDfxL4E7yBg"
            xmi:type="uml:Class">
              - <ownedAttribute name="enableScripting" xmi:id="_v0G9sPszEeeyDfxL4E7yBg"
                xmi:type="uml:Property" visibility="public">
                  <type xmi:type="uml:DataType"
                    href="pathmap://SysML14_LIBRARIES/SysML-Standard-
                    Library.uml#SysML.package_packagedElement_Libraries.package_pa
                  <defaultValue xmi:id="_60Bn4PszEeeyDfxL4E7yBg"
```

Figure 13: Example Excerpt from SysML file for Aeolus System

With this provided file, the HSF Translator Plugin needed to convert the SysML format into a structure that could be managed by HSF. The HSF file format is a much simpler XML representation of the system, as shown below. Unlike the SysML file however, the HSF file is much shorter. There is a concise declaration of each component, without any unnecessary information. The Aeolus example of this structure can be viewed in the figure below.

```
<?xml version="1.0"?>
<MODEL>
    <PYTHON enableScripting="true"> </PYTHON>
  - <ASSET assetName="Asset1">
      - <DynamicState ICs="[7378.137; 0.0; 0.0; 0.0; 6.02088; 4.215866]" DynamicStateType="PREDETERMINED_ECI">
            <EOMS EOMSType="orbital_EOMS"> </EOMS>
        </DynamicState>
        <SUBSYSTEM subsystemName="Access" Type="Access"> </SUBSYSTEM>
      - <SUBSYSTEM subsystemName="Adcs" Type="Adcs" slewRate="5">
            <IC value="[0.0; 0.0; 0.0]" key="ECI_Pointing_Vector(XYZ)" type="Matrix"/>
            <DEPENDENCY subsystemName="Access"/>
        </SUBSYSTEM>
      - <SUBSYSTEM subsystemName="EOSensor" Type="EOSensor" highQualityCaptureTime="7" midQualityCaptureTime="5"
        lowQualityCaptureTime="3" highQualityNumPixels="15000" midQualityNumPixels="10000" lowQualityNumPixels="5000">
            <IC value="0.0" key="numPixels" type="Double"/>
            <IC value="0.0" key="IncidenceAngle" type="Double"/>
            <IC value="0.0" key="EOSensorOn" type="Bool"/>
            <DEPENDENCY subsystemName="Adcs"/>
        </SUBSYSTEM>
      - <SUBSYSTEM subsystemName="SSDR" Type="Ssdr" bufferSize="5000">
            <IC value="0.0" key="DataBufferFillRatio" type="Double"/>
            <DEPENDENCY subsystemName="EOSensor" fcnName="SSDRfromEOSensor.asset1"/>
        </SUBSYSTEM>
      - <SUBSYSTEM subsystemName="Comm" Type="Comm">
            <IC value="0.0" key="DataRate(MB/s)" type="Double"/>
            <DEPENDENCY subsystemName="SSDR" fcnName="CommfromSSDR.asset1"/>
        </SUBSYSTEM>
      - <SUBSYSTEM subsystemName="Power" Type="Power" penumbraSolarPower="75" fullSolarPower="150" batterySize="1000000">
            <IC value="0.0" key="DepthofDischarge" type="Double"/>
            <IC value="0.0" key="SolarPanelPowerIn" type="Double"/>
            <DEPENDENCY subsystemName="Comm" fcnName="PowerfromComm.asset1"/>
            <DEPENDENCY subsystemName="Adcs" fcnName="PowerfromADCS.asset1"/>
            <DEPENDENCY subsystemName="EOSensor" fcnName="PowerfromEOSensor.asset1"/>
            <DEPENDENCY subsystemName="SSDR" fcnName="PowerfromSSDR.asset1"/>
        </SUBSYSTEM>
      - <CONSTRAINT subsystemName="Power" value="0.25" type="FAIL_IF_HIGHER" name="con1">
            <STATEVAR key="DepthofDischarge" type="Double"/>
        </CONSTRAINT>
      - <CONSTRAINT subsystemName="SSDR" value="0.7" type="FAIL_IF_HIGHER" name="con2">
            <STATEVAR key="DataBufferFillRatio" type="Double"/>
        </CONSTRAINT>
    </ASSET>
  + <ASSET assetName="Asset2">
    <REQUIREMENT value="20.0" type="LESS_THAN" name="imgcapqty"> </REQUIREMENT>
</MODEL>
```

Figure 14: Example HSF Input File for Aeolus System (Asset2 Collapsed for Clarity)

The output file for the HSF plugin is a typical input file for the Horizon Simulation Framework. The file cites subsystems, parameters, dependencies, and constraints, just like the SysML file, but in a much simpler format. At the highest level is the model tag, which contains and defines three major components: Python, the asset(s), and the requirement(s). The model starts by defining any parameters pertaining to Python scripting, as shown below.

```
<PYTHON enableScripting="true"> </PYTHON>
```

Figure 15: HSF Input - System Python Declaration

The next major component of the model is the asset, which is the physical system. This tag contains all of the details which belong to the system. Models have the capability to possess multiple asset children. This consists of the dynamic state, the subsystems, and the constraints. The dynamic state tag indicates the initial location and velocity of the asset at the start of the

simulation and the dynamic state type, which informs the Horizon Simulation Framework which reference frame coordinate system to use. The sub tag of the position callout is EOMS, which dictates which equations of motion the system will obey for the duration of the simulation. This parameter definition is crucial in calculating and updating asset(s) locations throughout HSF simulations. The figure below shows an example of an asset's dynamic state and its child, the equations of motion definition. [4]

```
- <DynamicState ICs="[7378.137; 0.0; 0.0; 0.0; 6.02088; 4.215866]"
  DynamicStateType="PREDETERMINED_ECI">
      <EOMS EOMSType="orbital_EOMS"> </EOMS>
  </DynamicState>
```

Figure 16: HSF Input – Dynamic State and EOMS Declaration

The asset is then broken down further into individual subsystems. The purpose of the subsystem is to specify certain features that belong to the physical asset. [4] Usually, subsystem operations and behavior are specified via scripted functions. These functions update the relevant state variables while HSF is executing the simulation. Subsystem dependencies can affect the operation of these scripted functions. Subsystems can be hard-coded, or non-scripted, into HSF but in order to effectively utilize subsystem modularity, it is recommended that the subsystems are scripted instead. [4] The following may be defined within a subsystem declaration:

- Subsystem Name
- Subsystem Type
- Subsystem Parameters

Subsystems may also possess child elements of their own that declare initial conditions of state variables or subsystem dependencies. Within the initial condition declaration, state variable's type, initial condition value, and name are defined. The only possible parameters within the dependency tag are the name of a previous subsystem and name of the Python-scripted function.

Each subsystem begins by defining the subsystem name, after which, any other relevant subsystem parameters are defined. The next level of subsystem definition is the initial condition. Each initial condition is established by a key, data value, and data type. The key is the parameter name of the initial condition.

Dependencies can also be defined as children of a subsystem. The role of the dependency function is to call out the function that the dependent subsystem will use as an access point. If Python scripting is enabled, a dependency may also define Python scripts to use during simulation. After defining any dependencies, the subsystem tag is closed and the next subsystem is opened for definition. This format continues until all subsystems are defined. [4]

```
<SUBSYSTEM subsystemName="Power" Type="Power" penumbraSolarPower="75"
fullSolarPower="150" batterySize="1000000">
    <IC value="0.0" key="DepthofDischarge" type="Double"/>
    <IC value="0.0" key="SolarPanelPowerIn" type="Double"/>
    <DEPENDENCY subsystemName="Comm" fcnName="PowerfromComm.asset1"/>
    <DEPENDENCY subsystemName="Adcs" fcnName="PowerfromADCS.asset1"/>
    <DEPENDENCY subsystemName="EOSensor" fcnName="PowerfromEOSensor.asset1"/>
    <DEPENDENCY subsystemName="SSDR" fcnName="PowerfromSSDR.asset1"/>
</SUBSYSTEM>
```

Figure 17: HSF Input – Subsystem, Initial Condition, and Dependency Declarations

After the completion of subsystem callout, the file format continues with the definition of subsystem constraints, calling out specific conditions in which the Horizon Simulation Framework should abort the current analysis and assess the next potential schedule. [4] The constraint callout defines the constraint type, value, name, and subsystem name. The type indicates the comparison argument, for a parameter value if it is higher, lower, or equal to a specified value. Should this argument be violated, HSF will abort the current schedule simulation and attempt a different schedule. [4] The value is the numerical limit of the constraint. The name is the string designation by which the constraint is referred. The subsystem name declares the subsystem in which the constrained parameter resides. The constraint tag then defines a child of its own: the state variable. Within the HSF input file, the state variable tag indicates the name and data type of the constrained parameter.

```
- <CONSTRAINT subsystemName="Power" value="0.25" type="FAIL_IF_HIGHER"
  name="con1">
      <STATEVAR key="DepthofDischarge" type="Double"/>
  </CONSTRAINT>
```

Figure 18: HSF Input – State Variable and Constraint Declarations

In the SysML output file, system components are effectively organized according to their role in the XML file. After declaring the extensive header, the SysML file defines the overall system name and model, which are classified under their respective “packagedElement” tags. The remainder of the system components follows a recognizable pattern. All nodes are defined through a “nestedClassifier” node. Any subnodes or children owned by that node are also established through the “nestedClassifier” tag. If any model node or subnode possesses a parameter, its name called out using an “ownedAttribute” designation. The value of the parameter is expressed under a “defaultValue” child of the “ownedAttribute”. An example of this SysML file structure is viewable in the image below.

```
- <ownedAttribute name="lowQualityNumPixels"
  xmi:id="_2lZRAPtDEeeyDfxL4E7yBg" xmi:type="uml:Property"
  visibility="public">
      <type xmi:type="uml:PrimitiveType"
          href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
      <defaultValue name="lowQualityNumPixels"
          xmi:id="_2lZRAftDEeeyDfxL4E7yBg" xmi:type="uml:LiteralString"
          value="5000"/>
  </ownedAttribute>
```

Figure 19: Example of SysML Output File Hierarchy

3.1.2 Plugin Logic and Implementation

As discussed in the previous section, the SysML file is full of information, some of which is irrelevant to the operation of HSF. Two elements are necessary in order to sift through such mountains of information: an understanding of the SysML file format, and the LINQ function library. The HSF translator plugin uses both to effectively process the SysML file.

The process of the translator plugin code begins by fetching the SysML output file. Knowing that the all-encompassing parent of the HSF model is the MODEL tag, the plugin immediately searches for the "packagedElement" tag using the literal "GetElementsByTagName" function and saves the XML node. This node contains a tree of all the information that is pertinent to producing the HSF model.

The saved XML node of system data then undergoes processing. Like the SysML file format, the HSF translator plugin abides by a similar hierarchy to establish major nodes and assign node parameters. As mentioned in the previous section, any major child node of a parent node is labeled as "nestedClassifier" within SysML. The "ownedAttribute" nodes are children of and directly refer to defining parameter names of "nestedClassifier" nodes. The "defaultValue" nodes are children of the "ownedAttribute" nodes and define the values of the stated parameters. This pattern is consistent throughout the SysML file, making the method of sorting data much more manageable. When a major node is detected, a check is first made for parameters. Then, a check is made for any subnodes. If a subnode is detected, a parameter check for the subnode is conducted. This process of looking for child nodes and their respective parameters repeats until the branch ends, at which point, the check steps back a level and searches for a node and respective parameters at the same level. When all of the subnodes and parameters have been located, the check steps back again, returning to finding nodes at the previous parent level. The process then starts anew with the next node, repeating until all of the modeled system components and their respective parameters have been translated into the HSF input file. This process is akin to HSF's schedule analyzer tool, which will step all the way through a schedule until a constraint is violated, at which point, it steps back to find alternate schedules. [4] The difference between the two, however, is that unlike the schedule analyzer tool, the translator plugin saves all paths because there is no constraint check, as all of the data is crucial in describing the model.

Another note to be made is that whenever a new node or attribute is detected, the related name and parameter information is immediately retrieved, saved, and appended to the HSF file

under construction. This “build-as-you-go” method helps ensure the HSF file structure remains suitable for reading during simulation and that all data is stored in the appropriate location with respect to its parent node. After the XML node data has been obtained and organized for use in HSF simulation, the new XML file is saved to the same directory and HSF is directed to use the newly generated file. The flowchart below depicts the procedure that the HSF translator plugin follows.

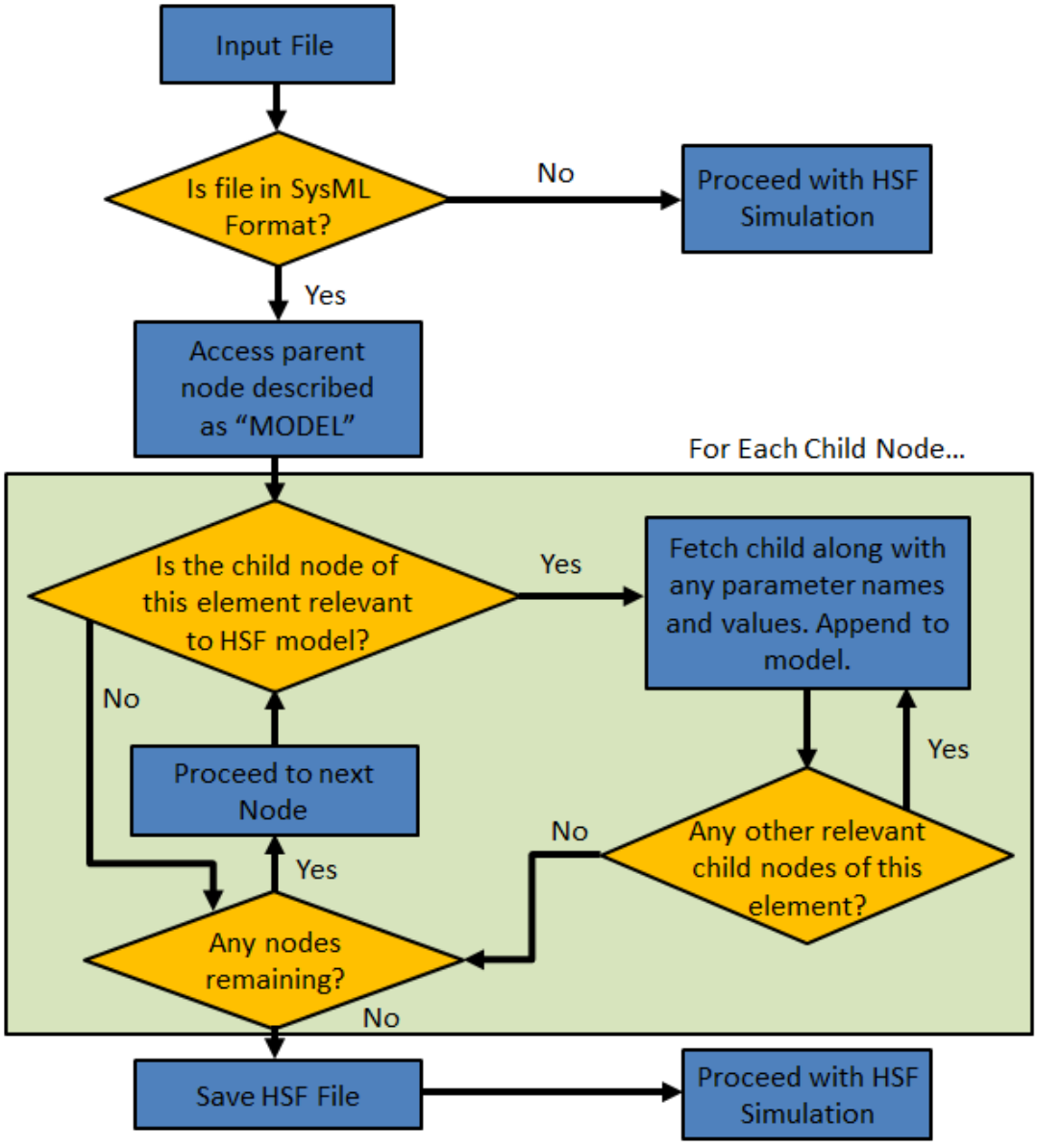

Figure 20: Flowchart of HSF Translator Plugin

By using functions supplied by the LINQ library, the code can retrieve all of the appropriate items without having to deal with the excessive information that is not pertinent to HSF. This is especially useful since output files from SysML can be extensive. After fetching a parameter and its respective value, the data set can appended to its parent node through the use of the LINQ toolset.

The C# LINQ toolset is used often in the Horizon Simulation Framework Plugin as it is known for its functions that are ideal for working with XML files. LINQ, or Language-Integrated Query, was introduced as a way to process large amounts of data of different formats. The intent of LINQ was to be a single processing source that could handle data that could differ in multiple dimensions. [7] In the case of SysML, the main dimensions of concern were volume, the size of the data package, and variety, structure of the data package. LINQ to XML is a series of tool within the library that are ideal for manipulating the data provided from SysML. One reason for this is because the toolset is centered on elements, the basis for the XML format. [2] Another reason for why LINQ to XML is an ideal toolset its simplicity of generating XML components. The library contains single-instruction functions that can populate XML elements and attributes quickly. [2]

This combination of programming in C# and using the LINQ library provided benefits from both tools. LINQ allowed for the manipulation of varying data structures of varying sizes. C# allowed for the structured use of “object-oriented programming”. [7] By taking advantage of these capabilities, the plugin could effectively meet its objectives.

The LINQ library’s XMLNodeList capabilities are a prime example of the power of the LINQ to XML toolset. The “GetElementsByTagName” function runs a query through the entire XML file looking a specific type of XML tag. Whenever the query correctly identifies the correct XML tag, the associated value is fetched for that instance. Conducting the query greatly simplifies the process of retrieving pertinent data as it sifts through and removes information that is already known to be irrelevant to the HSF model.

Aside from the fetching process, the Plugin used a few more LINQ tools to simplify HSF input file creation. XmlDocument was used to create a new XML file. Once the file was created, XmlNode was used to create tags and subtags, while XmlAttribute was used to add attribute names and values to the newly created tags. The AppendChild function allowed for continuous stacking of tags for subsystems and their subsequent initial conditions, constraints, and dependency functions.

3.2 System-Level Requirement Analysis Code

3.2.1 Requirements and Constraints

The current state of the Horizon Simulation Framework analyzes a system based on requirements placed at the subsystem level. A constraint is placed on a subsystem and HSF selects schedules in which these subsystem constraints are obeyed. It is easy for HSF to quickly conduct a subsystem constraint check since a value comparison can be made at each time step. One thing that HSF lacks, however, is the ability to analyze requirements at the system level. System-level requirements often require an investigation of the system as a summation of its parts. HSF, however, looks at the system one time step at a time. One of HSF's philosophies is that any state variable value at any given time step is independent of the value at any other time step. [4]

So how can an application that looks at subsystem state data at each time step analyze a scenario at the system level? In order to accomplish this, HSF must access data from the entire simulation. System-level requirements must be checked after simulation and schedule creation has completed. This also means that HSF cannot eliminate a schedule if the simulation fails to meet system-level requirements. Instead, when the post-simulation analysis is complete, HSF must generate an output indicating whether or not the requirements have been met.

To test the code, new variables had to be conceptualized that would be considered system-level parameters for the Aeolus simulation. The established criteria for a system-level parameter were that the parameter calculation must either depend on multiple time steps or rely

on data from multiple subsystems. As a result, the image capture quantity and data latency variables were created. Image capture quantity would represent the total number of targets captured in a schedule. Data latency would represent the average difference between the time an image was captured and when it was downlinked to the ground station. Both of these parameters required system-level analysis as they rely on data from multiple states within the simulation.

Adding system-level requirements into a model involved simple manipulation of the system input file. The <REQUIREMENT> XML node type was created. This node defines the parameter name, limiting value, and comparison argument for the requirement. The table below describes what each parameter signifies.

Table 4: Requirement Parameters and Descriptions

XML Parameter	Definition
Name	Parameter name. Helps HSF determine what calculations are required to analyze the requirement.
Type	Comparison argument. Indicates how to compare the simulation data calculation to defined requirement value. Can be any one of the following values: LESS_THAN, GREATER_THAN, LESS_THAN_OR_EQUAL, GREATER_THAN_OR_EQUAL, EQUAL, NOT_EQUAL.
Value	Limiting value. Represents the value to which HSF compares its simulation data calculation.

When the <REQUIREMENT> node was created, the question regarding the level of implementation arose. Would the tag be a child of the model, asset, or subsystem? The idea of adding the <REQUIREMENT> node as a child of a subsystem was quickly dismissed because system-level requirements could involve more than one subsystem. In addition, calling out a system-level requirement at a sub-system level would defeat the purpose of analyzing requirements at the system level. Calling out the <REQUIREMENT> node as a child of the <ASSET> tag was another feasible option since the system-level requirement would be at the same level of the subsystems. Once again, however, the name of "system-level" requirement implies that the statement is made at the system level, which encompasses all of the assets.

Additionally, defining system-level requirements would necessitate that the callout be repeated within all assets and add unneeded redundancy. Defining system-level requirements as children of the <MODEL> node proved to be the most useful since doing so would allow for analysis to be done at the highest level, validating all assets within the system without having to repeat the callout.

Because the analysis is conducted at the system level, requirement is defined at the same level as the assets. The requirement verification check is executed for all assets within the system, as there is the possibility that some assets within the system may fail to meet requirements, while others may succeed. An example of a <REQUIREMENT> node is shown in the figure below.

```
<REQUIREMENT value="20.0" type="LESS_THAN" name="imgcapqty"> </REQUIREMENT>
```

Figure 21: Aeolus Example of the HSF Model Requirement Node

3.2.2 Logic and Implementation

With the system-level requirements analyzer, the Horizon Simulation Framework runs a simulation as usual: analyzing the scenario, checking state data against subsystem constraints, and generating schedules. There are a few exceptions though. At the beginning of the simulation a requirement class is instantiated, which allows for system-level requirements to be analyzed later. During the XML model parsing process, HSF has also been updated to recognize the <REQUIREMENT> node. Each requirement node and its corresponding parameters are saved to a designated list. Upon completion of the simulation, HSF processes each requirement in the list. A function in the requirement class then parses the XML requirement node to fetch the requirement parameters and put them into a data structure of strings. The data structure is then run through another function in the class to execute the requirement analysis for the given inputs.

Throughout HSF, Python is used as a tool for scripting various components of the system, including the scheduler and subsystems. Since the capacity for utilizing Python already existed within the Horizon Simulation Framework, its implementation as a means for requirement

analysis would be relatively simple. The requirement class, just like the other HSF scripting classes, calls on scripting class libraries to access the capability to run Python.

When the class in HSF executes the function, it instantiates and accesses a function within the Python class coded specifically for analyzing the requirement. The inputs for the function are the same parameters defined in the <REQUIREMENT> node. There is an additional parameter included, which is the total number of assets. This is done so that the requirement verification can be conducted for each asset individually.

Upon execution, the Python function begins by fetching the relevant state variable output files. These csv files have been created as a result of the simulation and saved to the local drive. The files are processed and the data is consolidated into arrays. The arrays are then used in calculations to find the simulation's parameter value. Referring back to the Aeolus case shows examples of how the values are calculated for the two parameters.

Once calculation has completed, the value is checked against the comparison value provided in the input file. The comparison method is set from a series of if statements that will translate the comparison argument string into a symbol. Then the comparison analysis is executed, providing a true or false value depending on whether or not the comparison is correct. Upon completion of the check, the parameter name and comparison are printed to the generated text file. In addition, a statement is made on the validity of the comparison, indicating whether or not the system-level requirements were met. The entire process is then repeated for each asset within the system. After all requirement analysis has completed, the file is closed and the program ends its run.

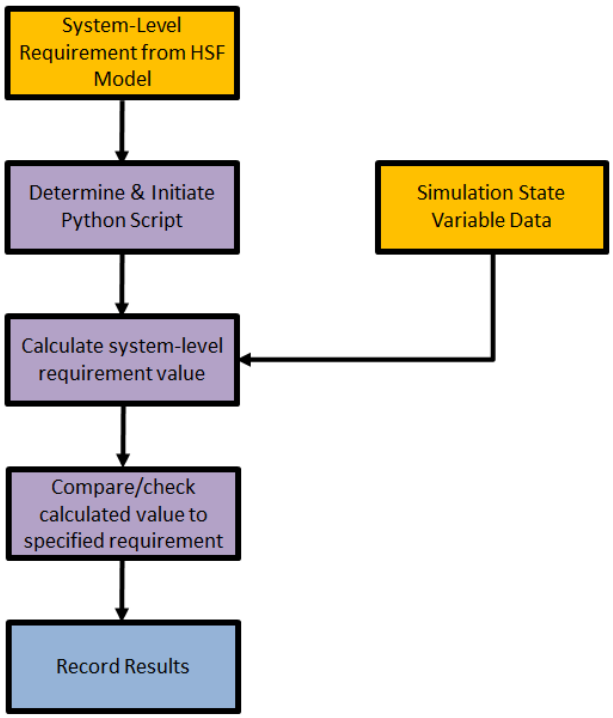

Figure 22: System –Level Requirement Analyzer Flowchart

As a supplement to the requirement analysis module, a template was created as a means of easily creating new requirements while maintaining consistency among analyses. Coding an entirely requirement evaluation from scratch can be a daunting task, so this tool helps alleviate unnecessary repetition. The template is a Python class file with the pre-existing base code and instructions to produce a new requirement. To begin, the user must save a copy under a new name, replacing any instances of the text "req_template" with the requirement name of their choice. Within the new Python file, users will need to add code to conduct the analysis and edit existing code to ensure that all relevant data is fetched and all requirement analysis is conducted correctly. There is caveat behind this template that can limit requirement creation. The analysis is based on the assumption that HSF is outputting the necessary state variables. If the analysis needs state variables that are nonexistent, the user will need to update the HSF codebase or scripted

subsystems such that the requisite state variables are recorded and saved to output files. A screenshot excerpt of the template is shown below.

```
class req_template(): #Replace "req_template" with Requirement Analysis Class Name
    def __init__(self): #DO NOT CHANGE
        pass
    def main(self, asset_num, comp_arg, comp_val_str):  # MAIN FUNCTION DECLARATION. DO NOT CHANGE
        comp_val = float(comp_val_str) #DO NOT CHANGE
        # INPUTS FOR THIS FUNCTION
        # asset_num = number of assets (integer)
        # comp_arg = comparison argument (string)
        # comp_val = comparison value (double)
        text_file = open("req_template.txt", "w") #Replace "req_template" with Requirement Analysis Class Name
        cwd = os.getcwd()
        #print(cwd)
        path = "C:/HorizonLog/Scratch"  # Data File Path per HSF
        os.chdir(path)  # Change Directory

        # Operator Designations. DO NOT CHANGE
        ops = {"<": operator.lt, ">": operator.gt, "<=": operator.le, ">=": operator.ge, "==": operator.eq,
               "!=": operator.ne}

        for index in range(asset_num):  # For each asset...
```

Figure 23: HSF Requirement Analysis Template Excerpt

3.3 Problems Encountered & Reasoning

One tool that HSF's requirement analyzer takes advantage of in its latest revision is the implementation of Python scripting. HSF accomplishes this through the use of IronPython. IronPython is a library that allows C# programming to execute Python files. Historically, HSF uses this tool for scripting purposes. With IronPython, HSF can script components of the simulation like the scheduler, subsystems, and equations of motion through user-generated Python files.

Since this Python capability is already in place, the system-level requirement analyzer could also take advantage of this tool to validate requirements. Instead of calculating post-simulation data within the C# code, Python allows the user to generate a requirement-analyzer outside of C#. This contributes to the modularity of HSF. If the requirement analysis was conducted solely within HSF, users would need to create a new function or class in C# every time a new system-level requirement was made. With IronPython, the requirement verification can be executed in Python. Users can create a simple Python class with its own functions. This modularity gives users the ability to quickly add, remove, switch out, and modify requirements

without making any changes to the HSF codebase. In addition, if the requirement analysis Python files are already coded up, then the only necessary change that the user must make resides in the input file.

One of the advantages of working with the SysML output file is that it is in XML format, the same format as the desired HSF input. However, sifting through the required information from the file proved to be a daunting task because of the excess information. Within some particular tags, there would be lines upon lines of machine language and/or program settings, all of which were irrelevant to HSF. An example of the irrelevant data can be seen highlighted in the figure below.

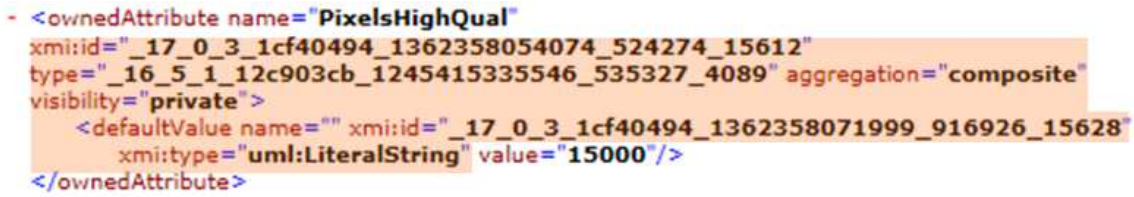

Figure 24: SysML Output Example of Unnecessary Data

In order to create an algorithm that would properly process the SysML file, a pattern had to be recognized such that the proper system parameters and constraints could be successfully retrieved. Because of proper formatting while defining the system in the SysML file, the pattern would be easily detected. By taking advantage of this pattern, the plugin code would be able to efficiently fetch the relevant parameters, without having to deal with the unwanted data that was only pertinent to the SysML format.

It was noted that names of parameters were stored under a common tag type called "ownedAttribute." Using the LINQ library "Descendants" function, the plugin was able to fetch all names under that XML descendant tag. The data were stored under the common attribute type, "name". Like the parameter names, parameter values we easily fetched using the same method. However, the parameter values were stored under the XML descendant tag called "defaultValue." For this data set, each item was stored under the attribute type, "value".

3.3.1 Code Choice

One crucial decision involved in writing the HSF translator plugin was deciding the programming language. Each language had its own set of advantages and disadvantages, and it was important to select the code language most appropriate for the task at hand.

Perl is a programming language used for manipulating extensive lines of text. Perl's capability of creating complex lock and key arrays is also ideal for managing subsystem relationships. A series of nested arrays can be created, containing each subsystem's parameters, constraints, and their respective values. This layout is comparable to XML's nested tag format, which would allow for easy parameter organization and manipulation. Unfortunately, Perl would require an additional suite of add-ons. Installing the add-ons would be a waste of space, ultimately slowing the program down.

JAVA was another potential candidate for coding the plugin, but it was quickly realized that it would generate more issues with integration into the general code. JAVA did have the advantage of dealing with UML, but since the Horizon Simulation Framework is written in C#, having some way to bridge the gap between coding languages would be an unnecessary hassle.

MATLAB is commonly used in the aerospace industry and academia. The language is very easy to use and does not need to define objects. However in this case, MATLAB isn't ideal for handling extensive lines of text. Manipulating large text files causes MATLAB to slow down. MATLAB's forte resides in data analysis and representation. MATLAB would serve a better purpose on the front end, running scenario simulations in HSF, or the back end, analyzing and visualizing the data from generated schedules. Manipulating the SysML files requires a faster approach.

C is one of the most basic programming languages. C++ and C# are both built upon C, which would make it easier to integrate than other programming languages. C also has file manipulation capabilities. However, in this context, C does not have the proper tools catered to

XML file creation and manipulation. Coding the plugin in C would be excessive, especially when there are other languages with the proper tools available to complete the task at hand.

Writing the HSF translator plugin in C++ would've been an ideal case, as the previous revisions of the Horizon Simulation Framework were written in C++, allowing for easier integration. However, HSF had since been updated to operate in C#, making the decision an obvious choice.

Ultimately, C# was the best choice for coding the HSF translator plugin. The plugin could easily integrate with HSF as it is written in C# as well. Implementing the plugin would not require downloading an additional suite of cross-functional add-ons. Another advantage of C# is that there is a library of functions that specifically allows for simple manipulation of XML files, minimizing compilation time. C# has sets of commands that can query an XML file for specific tags and access each tags specific attributes and properties. Using such commands would allow for easy SysML file conversion.

RESULTS AND ANALYSIS

4.1 HSF Translator Plugin

4.1.1 Comparison of HSF Input Files

Multiple performance metrics were necessary to test the efficacy of the Horizon Simulation Framework Plugin. The first and simplest testing method was conducted by a visual comparison of the translated HSF input file compared to the original. This metric alone was not sufficient in its reliability since the comparison was merely superficial, so another method would be required. To increase the fidelity of the requirements verification, the plugin-generated file would need to be run in a simulation. The generated output data would be analyzed, visualized, and compared to the output data from the original file. If both simulations produced the same results, a successful translation could be confirmed.

Once the SysML file was processed within the HSF plugin, the resulting output file was compared to its HSF input counterpart. From a simple, visual comparison, it could be concluded that both files are very similar with minimal differences between them. All of the subsystems are properly ordered and have all of the correct parameters.

When comparing the parameters of the plugin output and original HSF input files, all of the values and parameter types matched up exactly. Visual comparison confirmed that the plugin algorithm correctly transcribed the SysML output file.

The only major visual difference noticed between files was the format of XML element callouts within the translated file.

For the original HSF input file, an element begins with an open angle bracket “<” and the element name (i.e. SUBSYSTEM, REQUIREMENT, CONSTRAINT, etc.). With the tag still open, the associated parameters and values are called out. Once all of the necessary parameters have been specified, a closing angle bracket “>” is added. Between opening tags and closing tags, child nodes may be defined with tags of their own. A closing tag ends the element callout, which includes a forward slash followed by element name, all within another set of angle brackets. All

elements types, with the exception of initial conditions, dependencies, and state variables, follow this pattern.

The plugin-generated file managed to create the desired elements, but a different syntax was utilized. In the translated case where the element has sub-elements, a closing tag was included, just like the original file. However, any XML node that had no sub-nodes, the closing tag was non-existent. Instead the opening tag would end with a forward slash after its last parameter. An example of the difference is highlighted in the figure below.

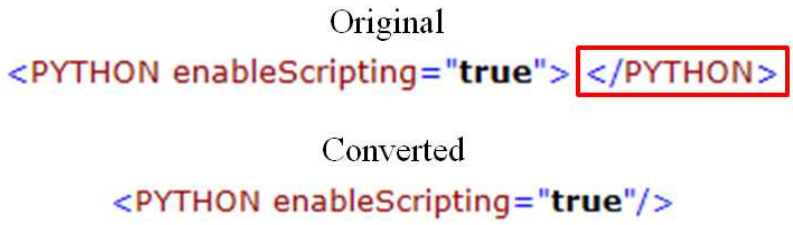

Figure 25: Example of Tag Closure Format Discrepancy

A discrepancy noted from translation, was the order of parameters within the SysML file affected order of appearance within the HSF file. The parameters would remain within the correct parent tag (SUBSYSTEM, CONSTRAINT, etc.). However, element parameters in the HSF file would be defined in the opposite order of their callout in SysML. Parameter order would not affect simulation output whatsoever, but for the sake of consistency, the sequence of parameter definitions were adjusted, simplifying the task of visually comparing XML files.

4.1.2 Comparison of Simulation Data

Despite the successful visual comparison between the two files, this alone could not provide adequate confirmation that the plugin output file can function as an acceptable HSF model. The file required further analysis via simulation in the Horizon Simulation Framework. With all other inputs being equal, the plugin output was processed in HSF as a replacement for

the original HSF input file. After running the simulation, the data was analyzed and compared to the original Aeolus system simulation data.

After running the simulation for the plugin-created system and comparing it to the original test case, it was apparent that all of the data output generated by HSF matched the data for the original Aeolus simulation. The simulation put out a series of files, providing state data for different parameters for the course of the simulation's time span. The data was compared by plotting the original Aeolus data against that generated from the SysML file translation. Because the simulation parameters called for the generation of two schedules, the state data of each original asset was compared to those belonging to its corresponding converted asset. From a visual analysis of each plot, it could be seen that both cases matched one another exactly. Upon further investigation of the raw data files, all of the data points matched, down to the time steps. This confirmed that the SysML translation was an accurate 1 to 1 portrayal of the original Aeolus system. This meant the HSF plugin had successfully translated the SysML into an HSF-readable format.

First the asset location for both cases was compared. For each recorded state, the X, Y, and Z positions were plotted with to the simulation time, as shown in the figure below. See APPENDIX A for graphs of the Y and Z values. For all three dimensions, each positional value of the converted case matched the positional value of the original case, given the same simulation time value.

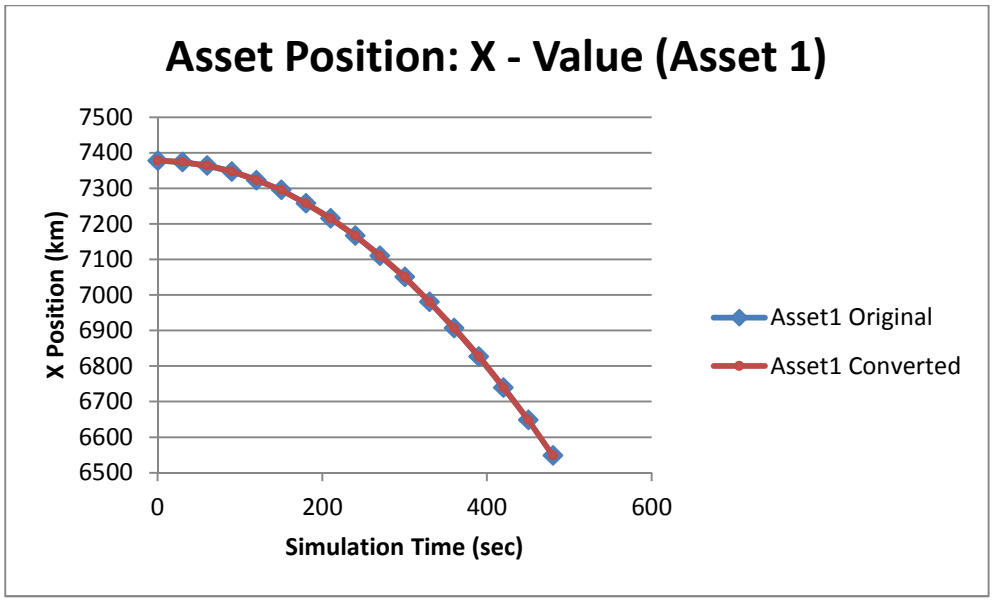

Figure 26: Original vs. Converted SysML Aeolus Simulation: X Position (Asset 1)

The same graphical check was conducted for the velocity and ECI pointing vectors. The results of these tests mimicked those of the positional vector. Both sets of data matched each other exactly. The figures below show the X-Value case, comparing the original and converted model data for velocity and ECI pointing.

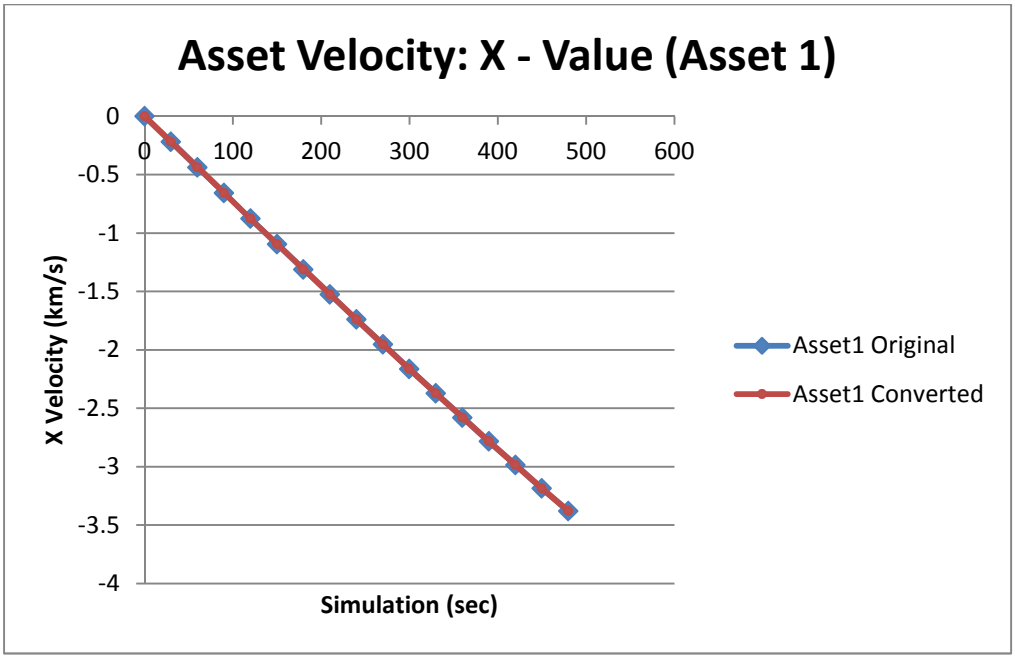

Figure 27: Original vs. Converted SysML Aeolus Simulation: X Velocity (Asset 1)

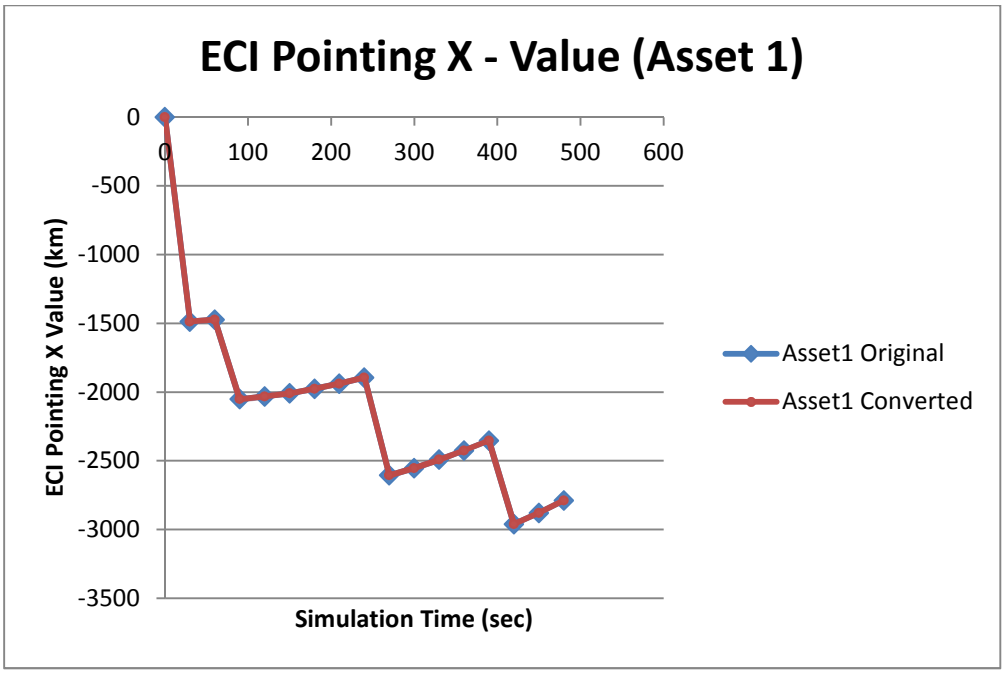

Figure 28: Original vs. Converted SysML Aeolus Simulation Data: X ECI Pointing (Asset 1)

In addition to position, velocity, and ECI pointing, state data regarding the Aeolus power subsystem was analyzed. The depth of discharge for both cases followed the same trend with the same data points. The value of the parameter followed a positive linear trend up to 0.01365 and exhibited a negative linear trend to 0, as shown in the figure below.

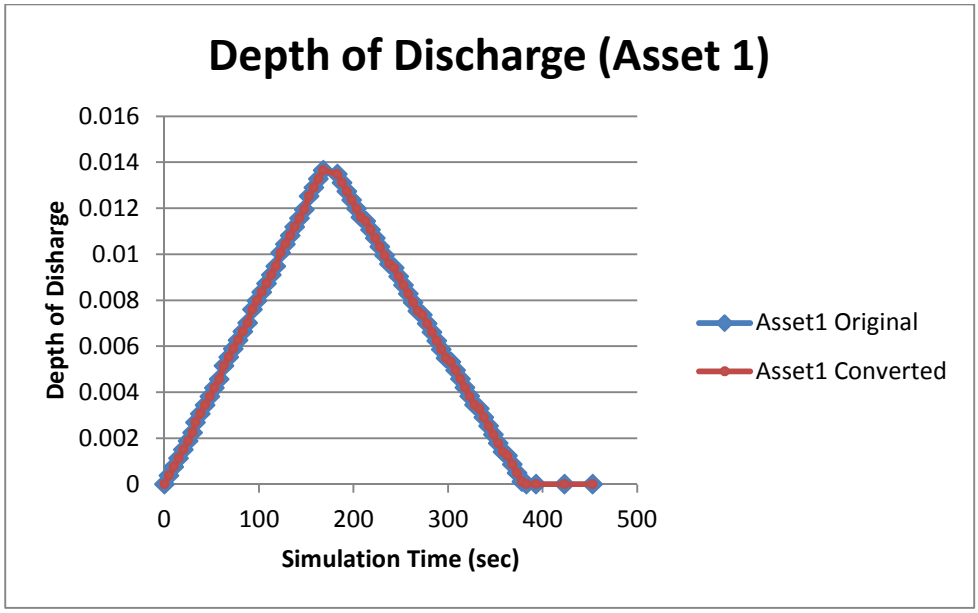

Figure 29: Original vs. Converted SysML Aeolus Simulation: Depth of Discharge (Asset 1)

The incoming solar power values for both data sets were also the same. The power value remained steady a 0 Watts for the first 153 seconds of the simulation, followed by an instant of 75 Watts while in penumbra, and generating 150 Watts for the remainder of the simulation. A visualization of this data can be viewed in the figure below.

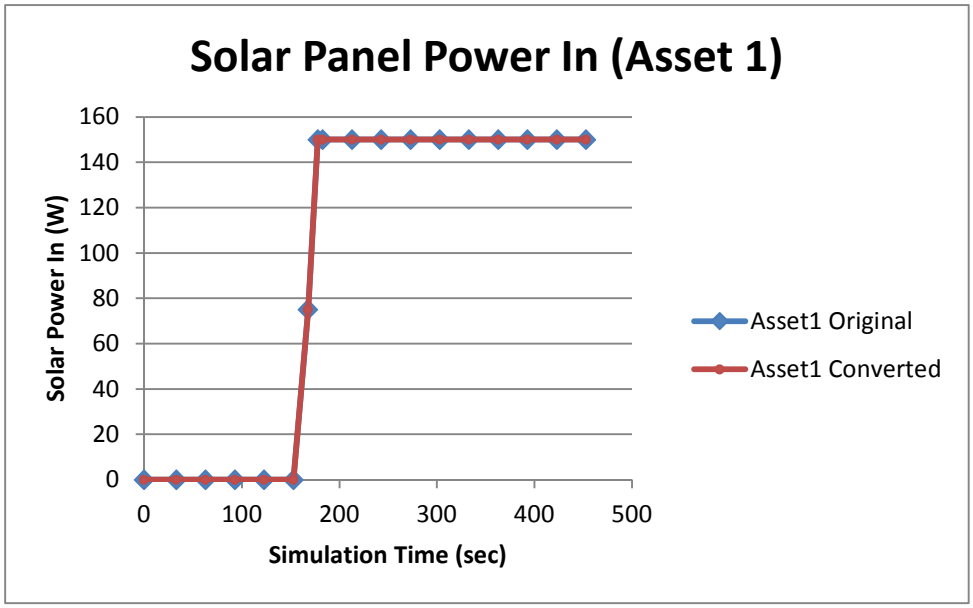

Figure 30: Original vs. Converted SysML Aeolus Simulation: Solar Panel Power In (Asset 1)

The data buffer fill ratio was another state data parameter that the simulation recorded. Once again, like the other parameters, the values for original and converted cases matched perfectly.

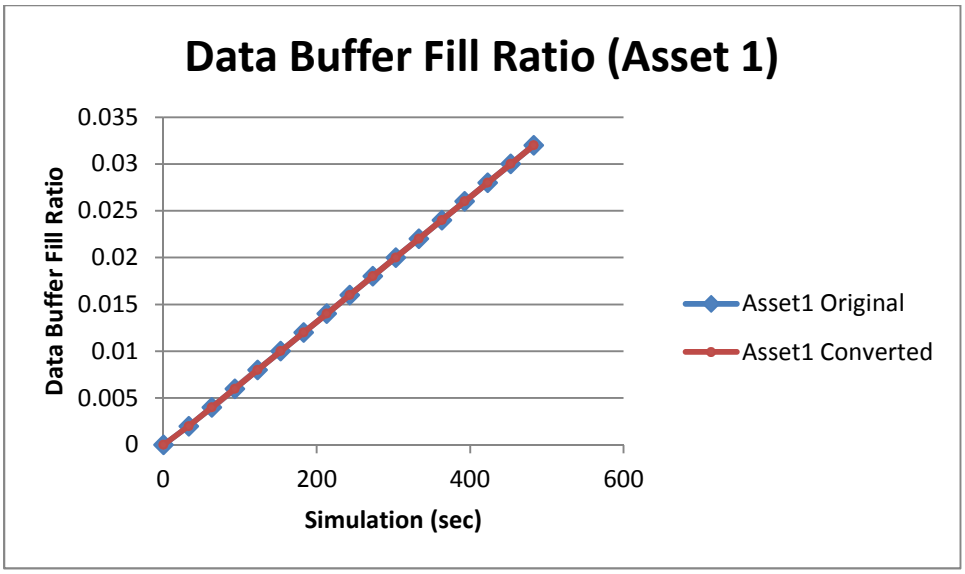

Figure 31: Original vs. Converted SysML Aeolus Simulation: Data Buffer Fill Ratio (Asset 1)

Simulation state data for the Earth-Observing sensor subsystem further solidified the pattern of matching data between the original and converted HSF system models. Plotting state data for spacecraft incidence angle, number of pixels captured, and the operating state of the sensor (on/off) against the simulation time, resulted in a repeating pattern for both cases. Data points varied between a zero-value and a max-value for the number of pixels and sensor operating state. The incidence angle peak values, however, slightly differed among one another in their own data set. The comparison between the original and converted cases still provided identical values as shown in the graphs below.

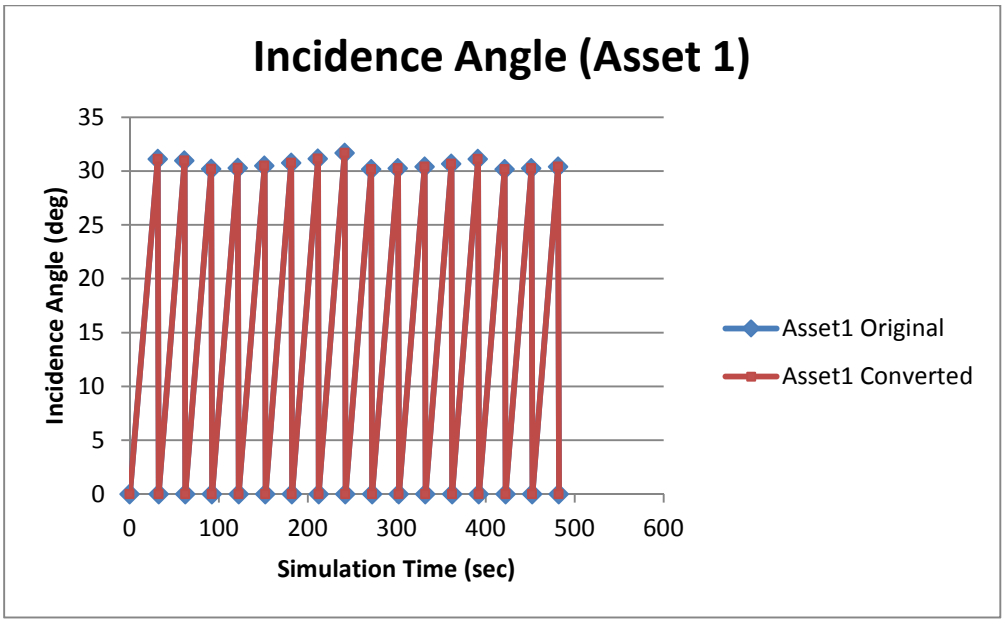

Figure 32: Original vs. Converted SysML Aeolus Simulation: Incidence Angle (Asset 1)

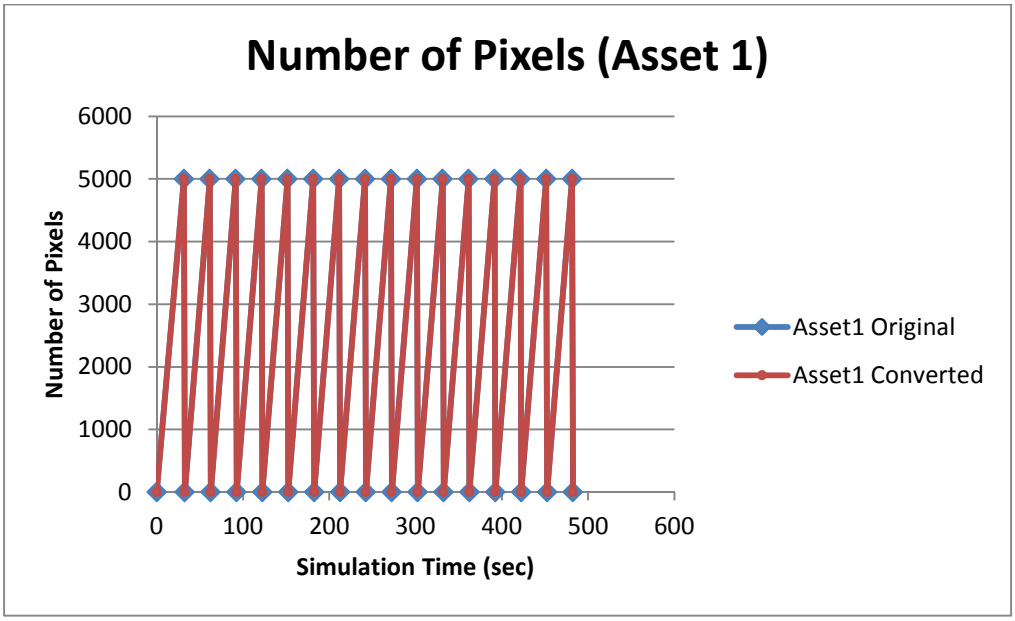

Figure 33: Original vs. Converted SysML Aeolus Simulation: Number of Pixels (Asset 1)

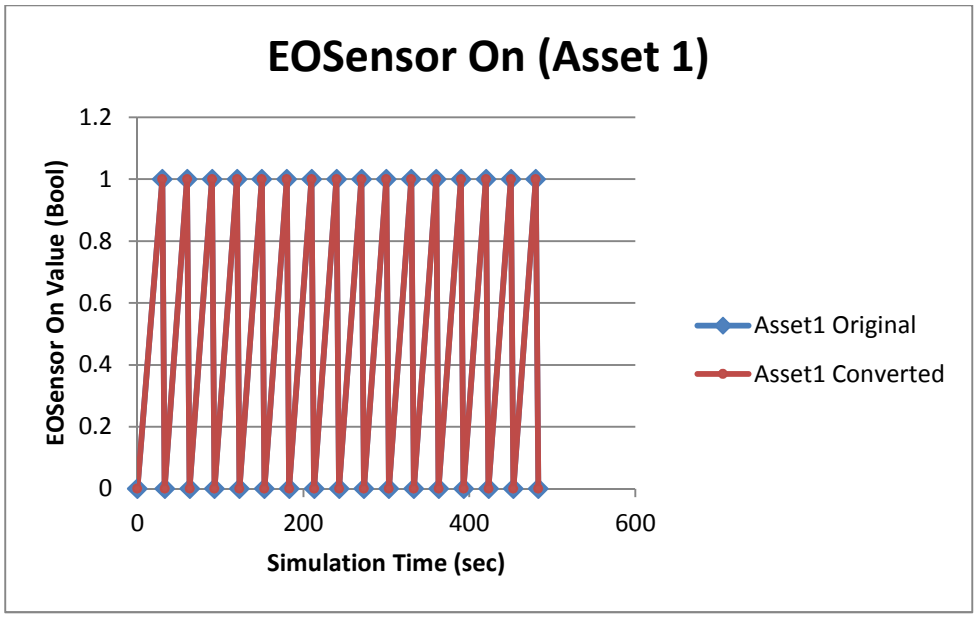

Figure 34: Original vs. Converted SysML Aeolus Simulation: EOSensor On (Asset 1)

As part of the output, the Aeolus simulations each produced a target state file. This itemized the schedule of captured targets, start and end times of the event, and the geographic coordinates of the target. Two series of data in each file contained strings, with one showing the names of the captured targets and the other showing the type of the captured target. To check this data, a string check was conducted comparing the target names and types from the original simulation to those of the converted case. A macro was used to compare each string pair and produced a Boolean output, indicating success or failure; displaying "TRUE" for an exact match and "FALSE" if the strings differed. All target name and type comparisons resulted in a "TRUE" output, once again confirming a match between both data sets.

All data set comparisons exhibited a common pattern: the original model simulation data matched the converted model simulation data. Numerical data matched down to the smallest significant figure (1e-10). If there were any deviation between the data sets, the error would need to be beyond the accuracy of the HSF's simulation data recording capabilities. Even after analyzing difference between the data sets, the results produced a series of zero values. Equation

1 below explains the calculation, where x_t is the translator simulation data, x_h is the original simulation data, and d is the difference between the two values.

$$d = x_t - x_h \tag{1}$$

Another item to note is that both the original and converted cases produced data sets with exactly the same amount of data points. When differences were calculated, the zero values confirm that the parameter values matched, but the length of the data sets paired with the zero values of the time data, further confirmed that the converted and original simulations matched. The figure below is an excerpt from the difference calculations in a Microsoft Excel spreadsheet.

Asset1 Error							
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
IncidenceAngle	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
numPixels	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
EOSensorOn	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
Asset2 Error							
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
IncidenceAngle	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
numPixels	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
Time	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
EOSensorOn	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00

Figure 35: Example Difference in Sim Data – Translated Case vs. Original (ADCS Data)

One final test was administered to prove a successful translation. Of the multiple outputs from the Aeolus simulation, one data set is a list that itemizes captured targets with the target name, type, and location, along with the start and end times of the task. A similar method was performed to evaluate the numerical data in this series. To compare strings like the target name and type, however, the "EXACT" macro within Microsoft Excel was a valuable analytical instrument. When comparing two strings, the macro produces a "TRUE" value if both inputs are identical and a "FALSE" value if there is any discrepancy, including letter case. The outcome of

the assessment once again proved that the simulations matched one another. Not only were the same system data values recorded at the same times steps, but the same mission targets were captured and the same instances, as shown in the figure below.

Asset1 Error/Check						
TargetName	TRUE	TRUE	TRUE	TRUE	TRUE	TR
TargetType	TRUE	TRUE	TRUE	TRUE	TRUE	TR
EventStartTime	0	0	0	0	0	
TaskStartTime	0	0	0	0	0	
TaskEndTime	0	0	0	0	0	
EventEndTime	0	0	0	0	0	
TargetLat	0.0000	0.0000	0.0000	0.0000	0.0000	
TargetLon	0.0000	0.0000	0.0000	0.0000	0.0000	
Asset2 Error/Check						
TargetName	TRUE	TRUE	TRUE	TRUE	TRUE	TR
TargetType	TRUE	TRUE	TRUE	TRUE	TRUE	TR
EventStartTime	0	0	0	0	0	
TaskStartTime	0	0	0	0	0	
TaskEndTime	0	0	0	0	0	
EventEndTime	0	0	0	0	0	
TargetLat	0.0000	0.0000	0.0000	0.0000	0.0000	
TargetLon	0.0000	0.0000	0.0000	0.0000	0.0000	

Figure 36: Example Difference in Sim Data – Translated Case vs. Original (Target Data)

The ultimate conclusion was that conversion of the SysML file into a format readable by the Horizon Simulation Framework was successful in meeting its objective. Comparing the plugin output with the original system model showed both files were very similar, with a few exceptions in sub-tag order. The simulation data further solidified the claim of success when both datasets provided the same values.

4.2 System-Level Requirement Analysis

Two test cases were created to test the effectiveness of the system-level requirement analyzer. The first case was a single system-level requirement. It involved a simple calculation to find the total number of images captured during the course of the simulation. The <REQUIREMENT> node of the HSF input file can be seen in the image in below.

```
<REQUIREMENT value="20.0" type="GREATER_THAN" name="imgcapqty"> </REQUIREMENT>
```

Figure 37: HSF Requirement Node – Image Capture Quantity (Greater Than)

For the quantity of images captured, the "EOSensor" state data was used. Whenever the Earth-observing sensor was on, a Boolean value of 1 was provided, indicating an instance when an image was captured. The number of instances was tallied up to calculate the total number of images captured. The equation below exhibits the calculation involved, where x_c is Boolean value indicating if the sensor was on, n is the total number of data points collected, and y_c represents the total number of images captured.

$$y_c = \sum_{i=1}^{n} x_c \tag{2}$$

After running HSF, the output in the form of a text file was created. The text file shows a four-line breakdown of the system-level requirement analysis as shown in the image below. Each requirement verification is organized by asset, as indicated by the first line. The second line shows the result of the calculation from the simulation data, which indicates that the first Aeolus asset captured 33 images. The third line is a statement of whether or not the system-level requirement was met. In this case, the requirement was satisfied, showing a success. The fourth line displays the comparison of the calculated parameter value to the user-defined value. The breakdown repeats a second time to show the analysis conducted for the second Aeolus asset.

```
ASSET 1
33 images captured
SUCCESS: SYSTEM-LEVEL REQUIREMENTS HAVE BEEN MET:
33 > 20
ASSET 2
33 images captured
SUCCESS: SYSTEM-LEVEL REQUIREMENTS HAVE BEEN MET:
33 > 20
```

Figure 38: Requirement Analysis Results – Image Capture Quantity (Success Case)

To confirm the accuracy of the simulation results, the analysis was conducted manually. The same calculation was made by manually accessing the state variable data file for

"numimgcapture". This state variable would store a true Boolean value every time the Earth-observing sensor turned on to capture an image. The total number of true Boolean values equaled thirty-three, confirming the accuracy of calculated value from the Python code. Comparing the calculated value to the "20" stated in the HSF file, the number of images captured during the simulation (33) was greater. This confirmed that the new module successfully validated the requirement.

This test only confirmed that the module correctly analyzed a case for successful requirement adherence. To test an unsuccessful case, the model input file had to be edited. The requirement type was changed from "GREATER_THAN" to "LESS_THAN". The <REQUIREMENT> tag was updated as shown below. If the requirement analyzer was coded properly, an adjustment to the type parameter would guarantee a failure statement.

```
<REQUIREMENT value="20.0" type="LESS_THAN" name="imgcapqty"> </REQUIREMENT>
```

Figure 39: HSF Requirement Node – Image Capture Quantity (Less Than)

HSF was run again, creating a new set of simulation data and an updated analysis of the requirement. The resulting text file is displayed in the figure below.

```
ASSET 1
33 images captured
FAILURE: SYSTEM-LEVEL REQUIREMENTS HAVE NOT BEEN MET:
33 is not < 20
ASSET 2
33 images captured
FAILURE: SYSTEM-LEVEL REQUIREMENTS HAVE NOT BEEN MET:
33 is not < 20
```

Figure 40: Requirement Analysis Results – Image Capture Quantity (Failure Case)

As expected, the output file indicated that the system did not meet the defined requirements. This can be seen on the third and fourth lines of each asset analysis. The third line states "FAILURE: SYSTEM-LEVEL REQUIREMENTS HAVE NOT BEEN MET" while the fourth line explains why. These two tests showed the operability of the system-level requirements analysis module in the success and failure cases.

The image capture quantity requirement proved that the module could successfully process requirements at the system level. This definition was quite simple in the sense that minimal calculation was involved. To examine the module's capability of handling complexity, a test requirement had to be formulated such that the relevant state variables resided in different subsystems and the calculations involved additional processing than just a simple summation. The result was a data latency requirement, measuring the average time between image capture and downlink to the ground station. The requirement callout in the HSF input file is shown in the figure below.

```
<REQUIREMENT value="100.0" type="GREATER_THAN" name="datalat"> </REQUIREMENT>
```

Figure 41: HSF Requirement Node – Data Latency

Calculating the data latency involved more complex calculations than the previous parameter. Investigating this requirement required accessing two state variable output files: the "databufferfillratio" state variable from the SSDR subsystem and the "numpixels" state variable from the EOSensor subsystem. After fetching the data from these two csv files, they were consolidated into arrays. Both datasets were then converted into bytes so that they could be processed equivalent using units. The array for the data buffer fill ratio was converted into the number of bytes by multiplying it by the size of the SSDR. The number of pixels was converted into bytes by multiplying the array by the conversion factor. Equations 3a and 3b explain the conversions where x_d represents the array of data buffer fill ratio values and x_p represents the array of number of pixels captured. The n_{SSDR} represents the conversion constant of the SSDR size (4098 bytes) and n_{pb} is the conversion constant of pixels to bytes (0.002 bytes per pixel). The resulting values are measured in bytes: the data buffer filled in (y_d) and data captured (y_p).

$$y_d = n_{SSDR} x_d \tag{3a}$$

$$y_p = n_{pb} x_p \tag{3b}$$

After the conversion, the total number of downlinked bytes was determined. A downlink was detected by searching the data buffer array for instances of decrease in value. The total amount of downlinked bytes was recorded.

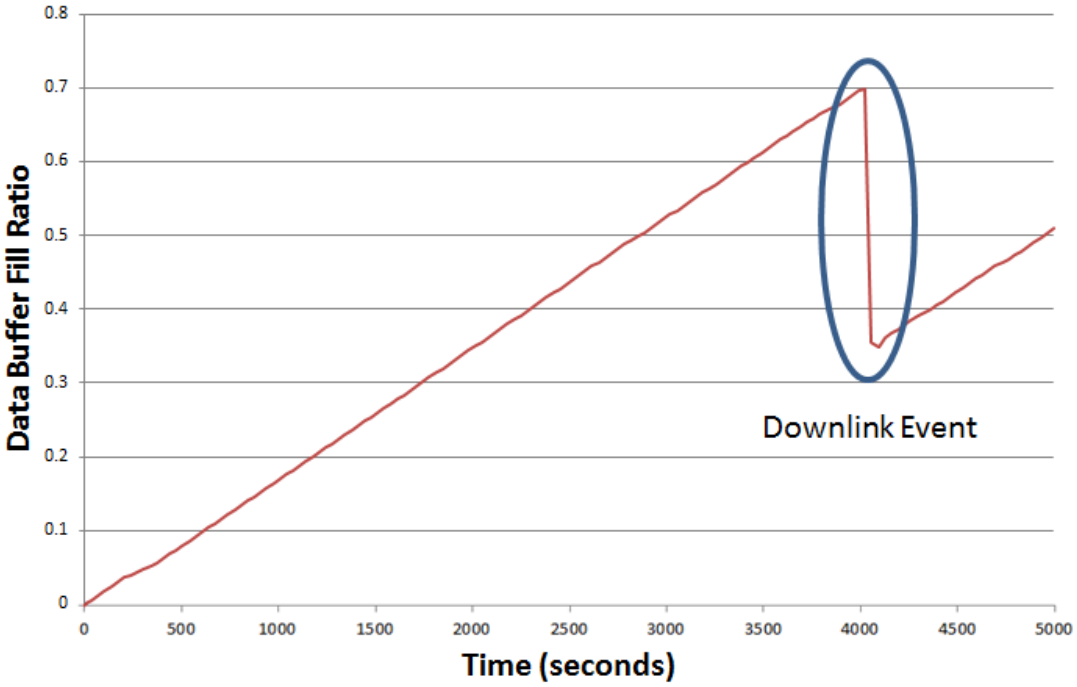

Figure 42: Data Downlink Detection

Working from the first time step in the simulation, the byte value was then used to determine how many captured images were downlinked. The data latency for each downlinked image was calculated by finding the difference between the times of image capture and the downlink times, as shown in Equation 4. Time of downlink and time of capture are represented by t_d and t_c, respectively, while Δt represents data latency.

$$\Delta t = t_d - t_c \tag{4}$$

The final value was achieved by averaging the array of data latency values, which in turn was compared to the requirement parameter definition. The formula for the calculation is shown in equation 4, where p is the number of data points and Δt is the average data latency.

$$\overline{\Delta t} = \frac{\sum_{i=1}^{p} \Delta t}{p} \tag{5}$$

In order to test the data latency requirement, a modification had to be made to the existing system model. The original Aeolus simulation had a time span that was too short to manifest any downlink event. Consequently, the simulation time span needed to be extended until a downlink was noticed. The result was updating a simulation input file by extending the simulation time from 1000 seconds to 5000 seconds.

The result of the simulation was approximately 3447.5 and 3387.7 seconds data latency for each asset respectively, which was greater than the stated value. Adherence to the requirement was confirmed, and the simulation provided the following output:

```
ASSET 1
Average Data Latency of 3447.541 seconds with 49 images downlinked
SUCCESS: SYSTEM-LEVEL REQUIREMENTS HAVE BEEN MET:
3447.541 > 100.000
ASSET 2
Average Data Latency of 3387.713 seconds with 47 images downlinked
SUCCESS: SYSTEM-LEVEL REQUIREMENTS HAVE BEEN MET:
3387.713 > 100.000
```

Figure 43: Results of Data Latency Requirement Analysis

After performing hand calculations for both Asset 1 and Asset 2 datasets, the final values were 3447.541 and 3387.713 seconds, respectively. Comparing the hand calculations to the HSF calculated values, authenticated an accurate analysis. Ultimately, the results of the test demonstrated that complex, cross-subsystem requirements could be analyzed. This success ultimately opened up a multitude of possibilities for the system-level requirements verification capabilities of the Horizon Simulation Framework.

CONCLUSION

5.1 Summary

The HSF Translator Plugin was proven to effectively convert a SysML model written in UML into a format catered to operate as an input for the Horizon Simulation Framework. With the help of a library of C# programming tools like LINQ to XML, the plugin code was able to extract simulation-pertinent data about the system. The plugin result further confirmed its success when it was entered as an HSF input and produced the same values as the original model. The first objective of creating a means of transforming a SysML model into an HSF model was accomplished.

The second objective was to provide the Horizon Simulation Framework a means of verifying requirements at the system level. Using the existing accommodations for Python scripting within HSF, a module was designed to parse in system-level requirements that were defined in the HSF model. The module processed the parsed requirement parameters, analyzed simulation data, and determined requirement adherence success or failure. Sample requirements had to be created to test the system, but the final result indicated the modules design and implementation were a success.

5.2 Strengths and Weaknesses

The main strength of the HSF translator plugin is its connectivity to SysML. Model representation in a different medium opens up possibilities and advantages in terms of system design. Each model type is ideal for different design purposes. SysML is effective at portraying the system and the relationships between its subsystems. HSF is ideal for simulating system operations and ensuring that it meets subsystem constraints. When used together, systems can be developed that factor in design considerations flowed down from both models. Consequently, the translator plugin expands the scope of operation for the Horizon Simulation Framework.

A model is only as good as what goes into it. Model fidelity is both a source of strength and a weakness for the translator plugin. Just as a well-constructed model will yield good results,

a poorly constructed model will provide poor results. This falls in line with the "garbage in, garbage out" philosophy of HSF. Not every SysML model placed into the plugin will provide a model that generates a successful HSF output. One reason for this is because a system may be successfully modeled in SysML, but it may not meet the established constraints. As a result, HSF will fail to generate a working schedule. This failure may not necessarily mean the system is poorly modeled, but further investigation may be necessary in adjusting subsystem parameters or constraints. The same concept applies to the system-level requirement analysis module. A requirement can be effectively defined and encompass exactly how the user wants the system to operate, but it does not guarantee that the designed system will meet the requirement. However, this does indicate that the responsibility of proper requirement construction falls upon the user. If the systems engineer wants to determine how well a system adheres to a requirement, then they must understand the requirement and possess the skills to appropriately portray it in the model.

Another weakness of this translator plugin is that it can only translate input files that pertain to the system parameters. As stated earlier, one useful advantage of the Horizon Simulation Framework is its modularity, where users can swap out subsystem functions as needed. The translator plugin, however, does not possess the capability to translate SysML models into subsystem functions coded in C#. If a user desires to test a system with different subsystem functions, the functions must be manually created by the user through C# or scripting in Python.

The system-level requirement analyzer also has strength in its modularity. A system-level analysis can be easily coded up in Python. After creating the Python code, users need only to update the system input file to ensure that the requirement is being called out. This process is repeatable for as many requirements the user can formulate. There is no need to make adjustments to the core code within HSF and rebuild every time a requirement changes. Modularity in the analyzer allows HSF to test a variety of requirements without making a change to the code base.

If subsystem-level constraints are defined in an HSF model, the simulation will ignore schedules that fail to comply with the specified constraints. Only schedules that adhere to the constraints will be selected and shown. The system-level requirement analyzer, however, does not behave in the same manner. The system-level requirement analyzer does not indicate requirement failure during simulation. Only after the simulation is complete does the module notify the user of the results. It will not skip schedules that fail to meet system level requirements. The end result for the user, however, will be similar. If HSF cannot generate a schedule that follows subsystem constraints or system requirements, the user must reassess the model and make the necessary changes to close on a working system design.

As its name states, the system-level requirement analyzer gives HSF the ability to look at the system as a whole. The previous version of HSF only constrained parameters at the subsystem level. This new ability broadens the scope of the Horizon Simulation Framework. Interdisciplinary requirements can be formulated and studied, giving further insight into how different subsystem parameters work together to impact higher-level system operations. Even if parameters operate independently within their respective subsystems, they can affect the system mission as defined in the system-level requirement. Another aspect of HSF is the independence of each state. In a simulation, any given state behaves independently of any other state. The system-level requirement analyzer broadens the scope of HSF again and gives user a look at the entire system as a culmination of states. Operators can define requirements that evaluate state variables at all time steps within the simulation. This new module gives system engineers using HSF the opportunity to take a step back and view the system as a collective of related subsystems operating over a period of time, instead of a series of limited state variables.

5.3 Lessons Learned

As stated previously in the strengths and weaknesses, the quality of requirements is dependent on the user. HSF and SysML are merely tools used in representing system. The

system-level requirement analyzer can determine whether or not requirements will be met, but the quality and depth of the requirement definitions are up to the user. This is why there were two different requirement parameters that were tested. The image capture quantity was a simple parameter dependent on a single state variable but span across multiple time steps. The data latency was much more complex and required data from multiple state variables originating from different subsystems. The intricacy of defined requirements relies upon the ingenuity and the higher-level thinking of the person operating the tool.

Another lesson learned is that modularity is a powerful tool in model-based systems engineering. Giving system engineering tools the potential to interchange working pieces generates a capacity to test multiple system configurations without the excessive upstart time to conduct a complete redesign every time a system aspect is tweaked. This lesson was learned while designing the system level requirement analyzer. The original plan was to hard-code the single requirement analysis within the Horizon Simulation Framework. This would have proven to be a daunting task since there are a near infinite number of possible requirements. Coding and testing each possible analysis would have taken an excessive amount of time. Instead, the existing Python scripting infrastructure was utilized to create a module where requirements could be interchangeable. This implementation reduced the amount of module testing to its ability to process the Python requirement analysis script. In addition, it placed more control into the hands of the user. The user could invent their own requirements, construct the analysis in Python, update the HSF input file, and execute the simulation. The fact that Python was the scripting tool further increased the “user-friendliness” of the module as it is a much simpler coding language. By focusing on modularity, the overall HSF experience was improved greatly in terms of capability and ease of use.

One lesson learned is that an understanding the system immensely helps in generating system level requirements. For any system-level requirement parameter, one must know which state variables will be needed in its calculation. Going one step further, one must also know into

which subsystems each state variable must tree. Some requirement parameters will necessitate the creation of new state variables, making it imperative to recognize which state variables are housed within which subsystems. The first test of the system-level requirement analyzer was extremely simple in the sense that it only required one state variable, “EOSensorOn”, to evaluate the requirement. The sole calculation involved was summing up the total number instances in which the Aeolus imaging sensor turned on to capture an image. Though the requirement was useful in proving the module’s operability, it lacked the complexity to exhibit how different subsystems interacted. Thus, the data latency requirement was established as means to exhibit an interaction between subsystems while simultaneously validating a much more complex requirement. In order to properly test this requirement, the origins of the parameter had to be derived. The “numpixels” and “databufferfillratio” state variable values were essential in calculating the data latency parameter. These state variables belonged to the “EOSensor” and “SSDR” subsystems respectively. Understanding the origins of these state variables provided greater insight into formulating and analyzing the system-level requirement.

5.4 Future Work

5.4.1 Editing and Creating HSF Equation Functions

The plugin can only do so much in terms of simple system modeling. It is solely designated for converting SysML files into HSF input files. The plugin does not affect the HSF code itself, which means it cannot make changes to the subsystem functions within the HSF source code. The next step in increasing the cross-functionality between SysML and HSF would be to create another plugin that would cater to creating and/or editing subsystem functions. Users would first model a single subsystem in SysML with default parameters and equations. Theoretically, the new subsystem plugin would read in the file, identify the parameters and equations, and then create a function for the source code. However, because all of the source code functions are linked and reference one another, it would be a daunting task to comprehensively integrate the function into HSF. A more realistic approach would involve editing existing

functions. The fetched parameters and equations would replace the counterparts found in the subsystem function. Doing this would allow users to experiment with a greater range of parameters that were originally unavailable to them in the HSF input files.

5.4.2 Integration into PLM Software

Conceptual system development is only one part of the design process. The entire Horizon Simulation Framework could be integrated into a larger product life management system. This would allow users to design initial system concepts and save relevant files into a larger database. Other users could then access this data to further refine analyses of their subsystems and offer higher fidelity results. Files in this database could be distributed to multiple users in a review process to ensure that all groups have viewed and approved of certain system design decisions. The creation and integration of HSF into such a software suite would make the entire design process much more similar to that of actual industry.

5.4.3 Expansion of Model Definitions

The current make-up of the typical HSF model consists of subsystem and their respective parameters, constraints, and initial conditions. In the engineering industry, the ultimate goal of model-based systems engineering is to create a single model for all aspects of the design process. Adding more functionality to HSF would help accomplish this goal. This could include, but is not limited to integration of CAD models, thermal meshes, and trajectory visualization. The task would involve coding a series of software suites that would provide the addition functions. Alternatively, interface code could be created, allowing HSF to “talk” to already existing modeling software, like PTC CREO, STK, and MatLab.

5.4.4 System-Level Requirement Library

The system-level requirement analyzer is a modular tool in which Python files evaluate user defined requirements. A potential project for future development would be a series of system-level requirements. The project would involve formulating requirements and understanding which state variables are necessary for their evaluation. If new state variables are

needed, the existing subsystem classes within HSF would be updated to produce them. Then, a class would be coded in Python for each new requirement. With enough Python files, students can create an entire library of requirements. Future HSF users would be able to make arbitrary selections from a long list of pre-coded system-level requirements.

BIBLIOGRAPHY

[1] Butler, Brian, “Dynamic Model Creation and Scripting Support in the Horizon Simulation Framework” California Polytechnic State University, San Luis Obispo, 2012.

[2] S. Eichert, F. Marguerie, and J. Wooley, “LINQ in Action,” Dreamtech Press, 2008.

[3] S. Friedenthal, A. Moore and R. Steiner, “A Practical Guide to SysML: The Systems Modeling Language,” Burlington, MA: Morgan Kaufmann Publishers, 2008.

[4] Kirkpatrick, Brian, “‘Picasso’ Interface for Horizon Simulation Framework,” California Polytechnic State University, San Luis Obispo, 2010.

[5] Luther, Shaun, SysML Based CubeSat Model Design and Integration with the Horizon Simulation Framework, California Polytechnic State University, San Luis Obispo, 2016.

[6] E. Mehiel, "Model Based Systems Engineering," Cal Poly, San Luis Obispo, 2012.

[7] Meijer, Eric, “The World According to LINQ,” Communications of the ACM (Volume 54. Issue 10), New York, NY, 2011.

[8] O’Connor, Cory M., “Horizon: A System Modeling and Simulation Framework for Systems Engineering Utility Analysis,” California Polytechnic State University, San Luis Obispo, 2009.

[9] J. Wolfrom, "Model-Based Systems Engineering (MBSE) Overview," Applied Physics Lab.

[10] E. Andrade, P. Maciel, G. Callou, B. Nogueira, “A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints,” Federal University of Pernambuco, Recife, PE, Brazil, 2009.

[11] “What is an Embedded System? - Definition from Techopedia.” Techopedia.com, Techopedia Inc., 2018, www.techopedia.com/definition/3636/embedded-system.

APPENDICES

Appendix A. Additional Horizon Simulation Framework Graphical Data Visualizations

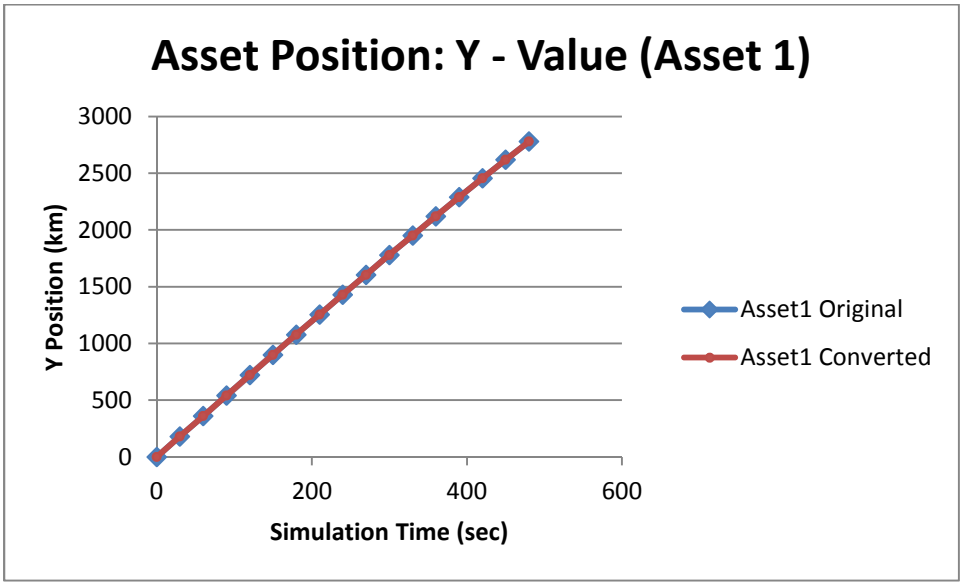

Figure 44: Original vs. Converted SysML Aeolus Simulation: Y Position (Asset 1)

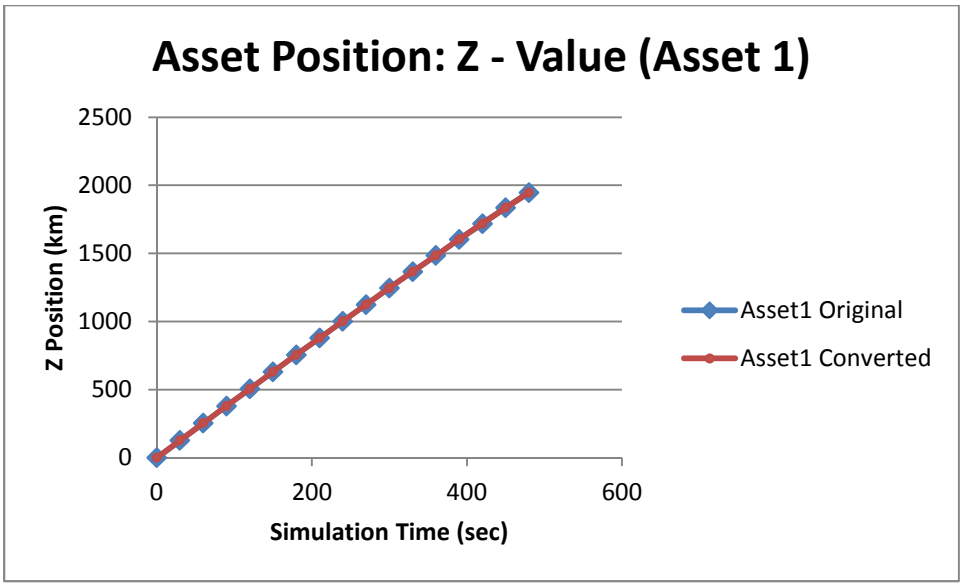

Figure 45: Original vs. Converted SysML Aeolus Simulation: Z Position (Asset 1)

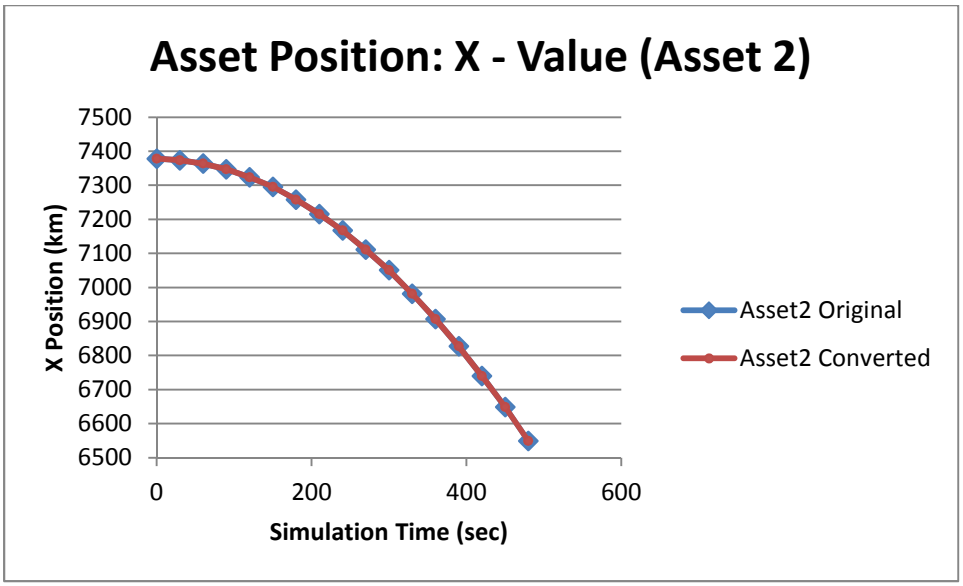

Figure 46: Original vs. Converted SysML Aeolus Simulation: X Position (Asset 2)

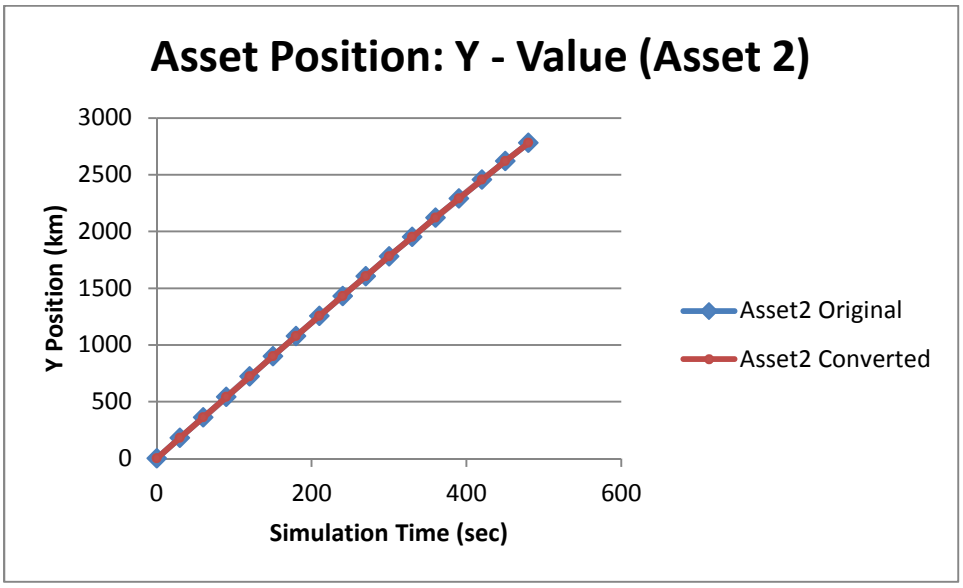

Figure 47: Original vs. Converted SysML Aeolus Simulation: Y Position (Asset 2)

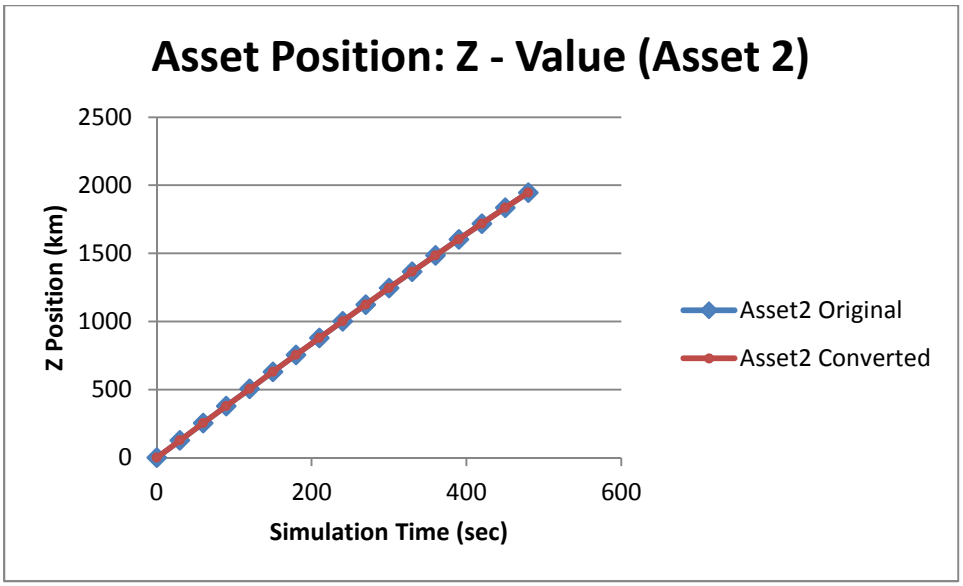

Figure 48: Original vs. Converted SysML Aeolus Simulation: Z Position (Asset 2)

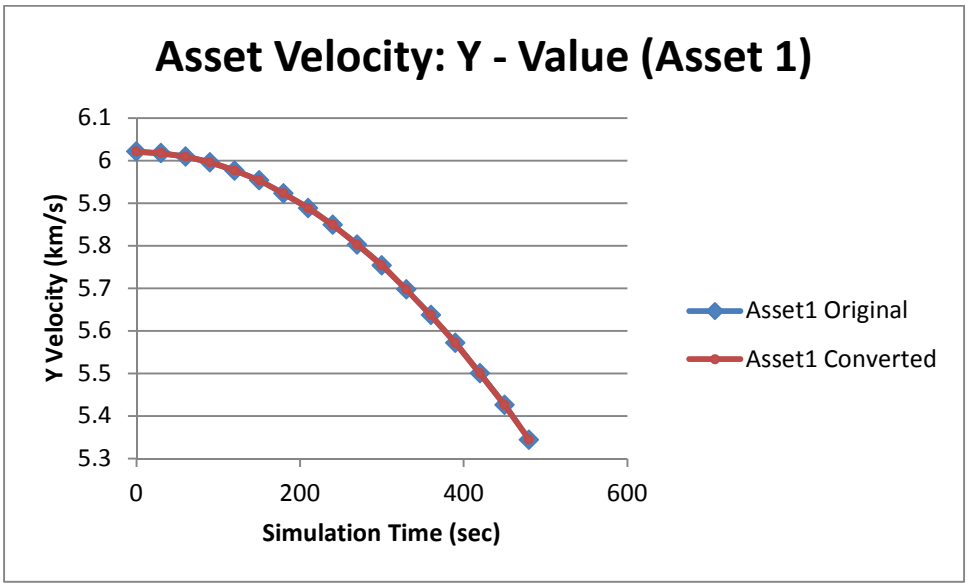

Figure 49: Original vs. Converted SysML Aeolus Simulation: Y Velocity (Asset 1)

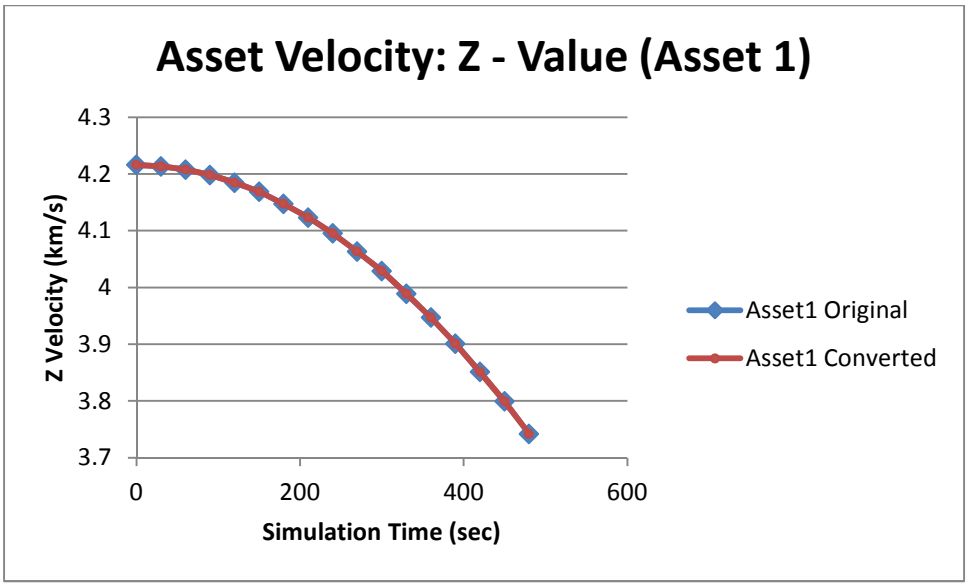

Figure 50: Original vs. Converted SysML Aeolus Simulation: Z Velocity (Asset 1)

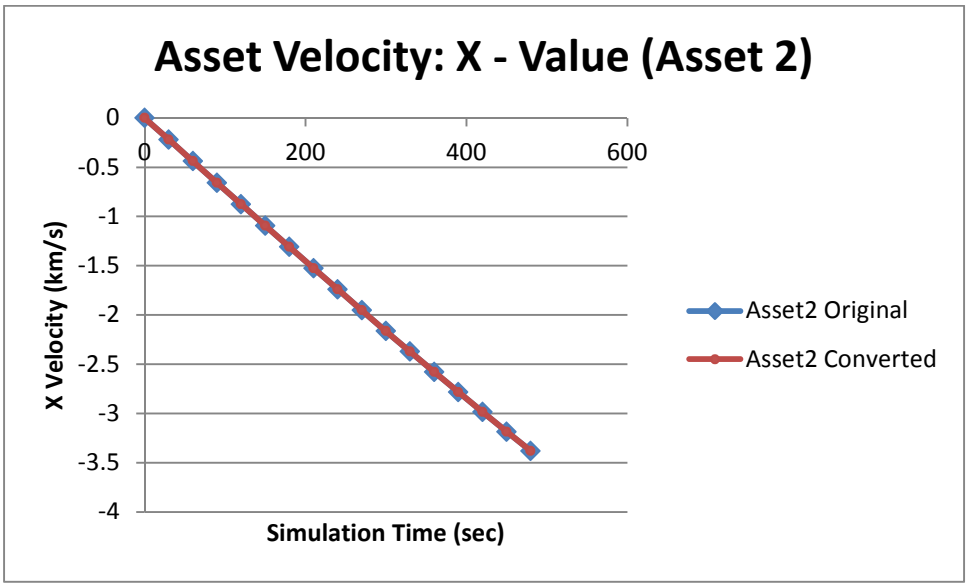

Figure 51: Original vs. Converted SysML Aeolus Simulation: X Velocity (Asset 2)

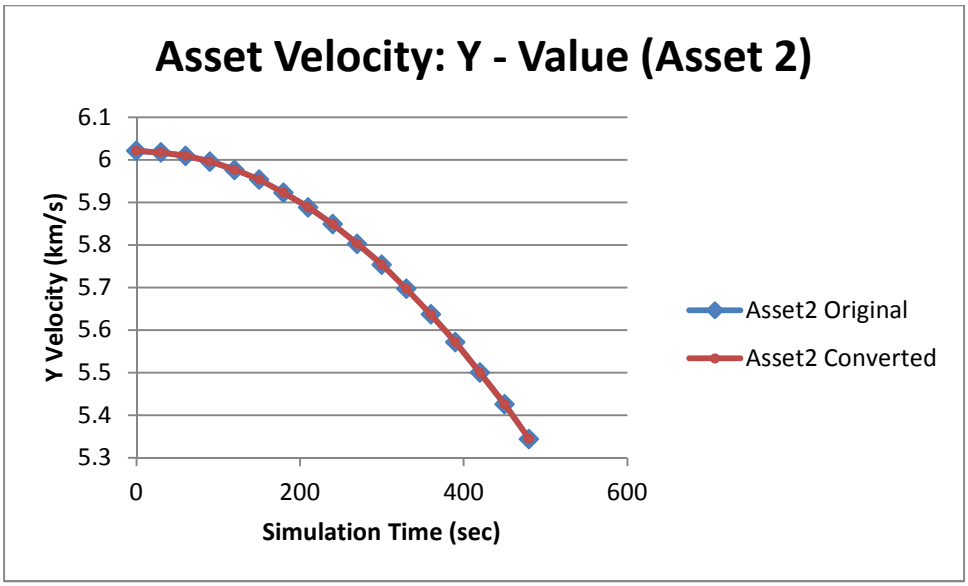

Figure 52: Original vs. Converted SysML Aeolus Simulation: Y Velocity (Asset 2)

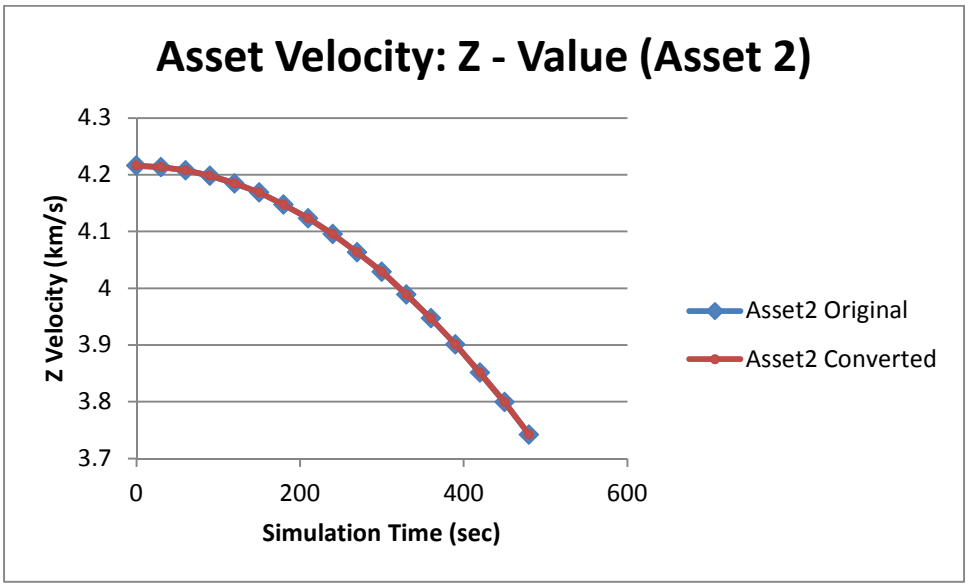

Figure 53: Original vs. Converted SysML Aeolus Simulation: Z Velocity (Asset 2)

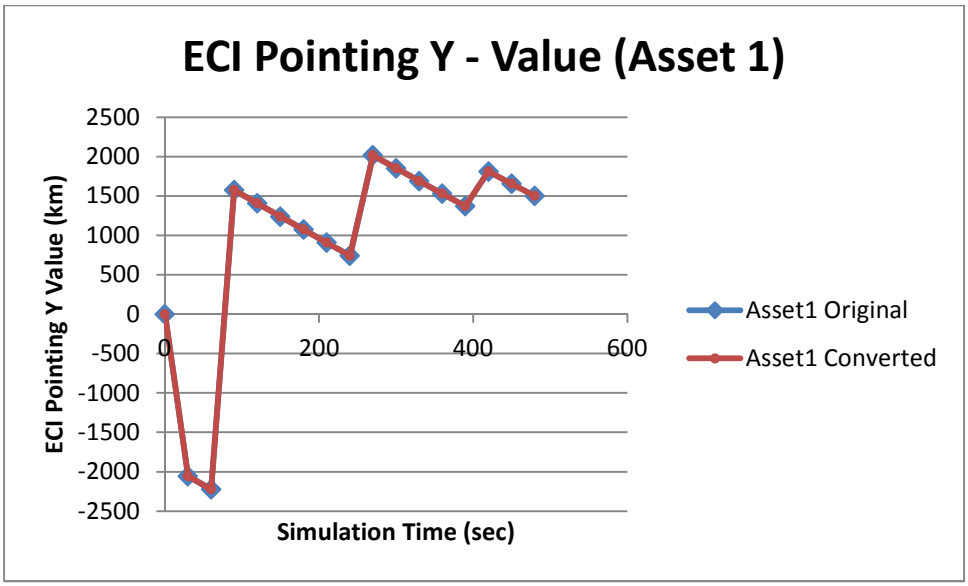

Figure 54: Original vs. Converted SysML Aeolus Simulation: X ECI Pointing (Asset 1)

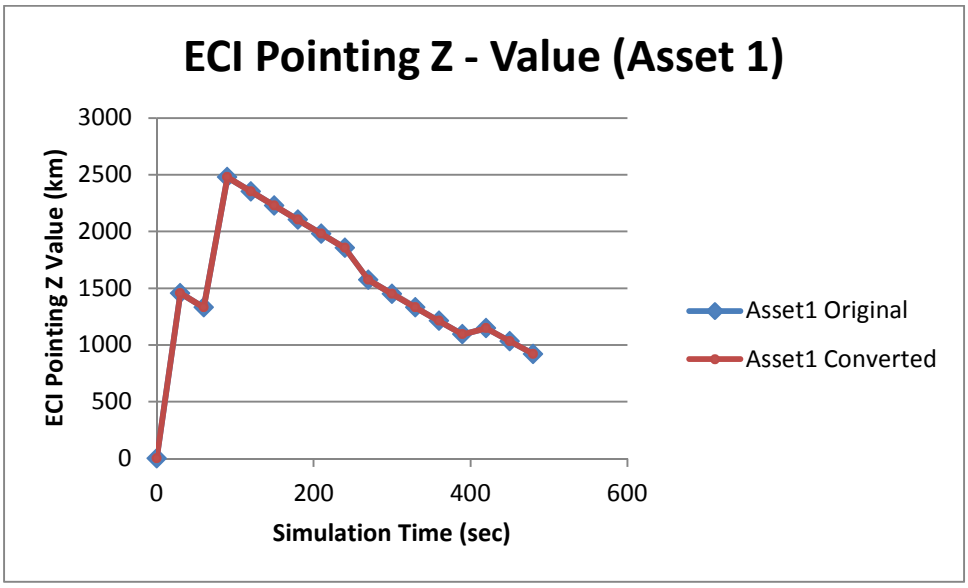

Figure 55: Original vs. Converted SysML Aeolus Simulation: Z ECI Pointing (Asset 1)

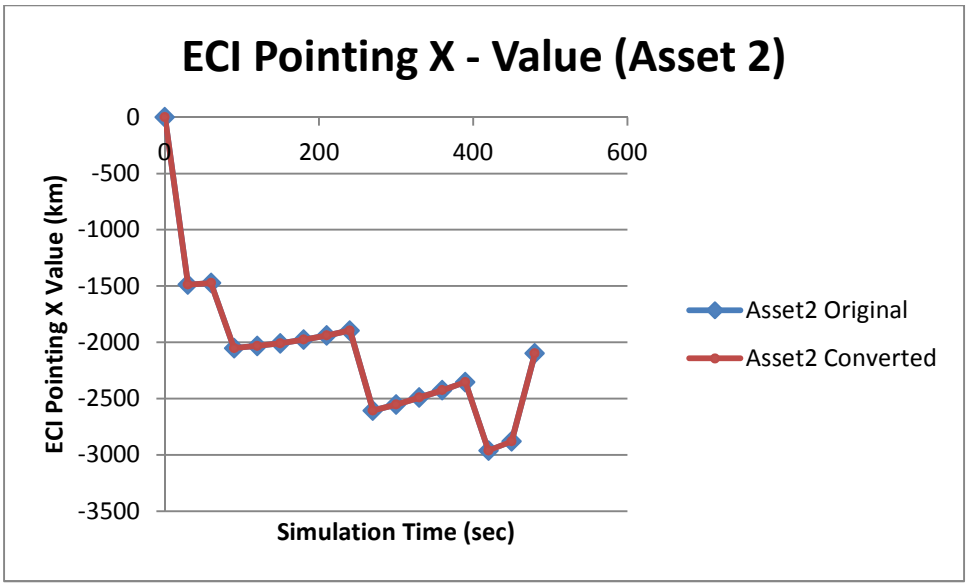

Figure 56: Original vs. Converted SysML Aeolus Simulation: X ECI Pointing (Asset 2)

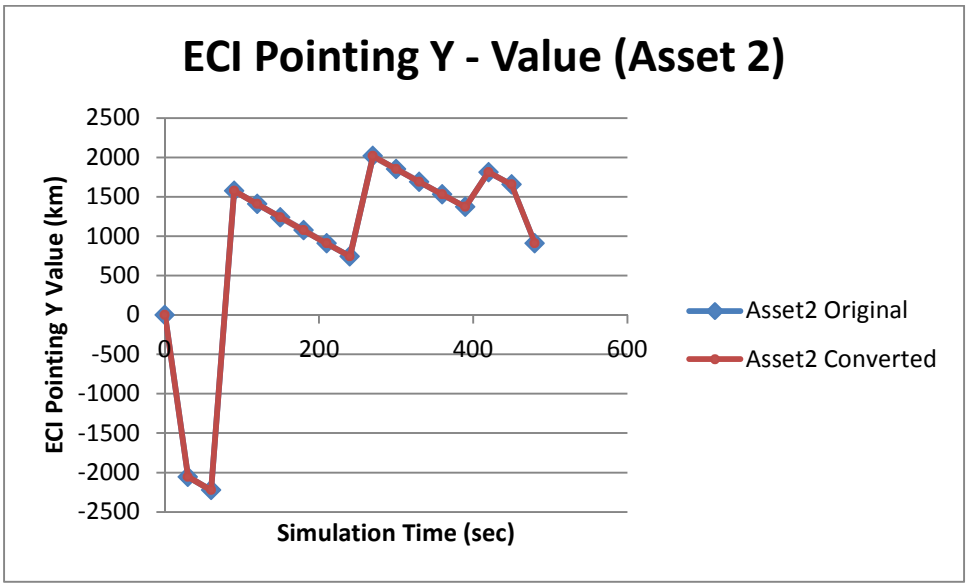

Figure 57: Original vs. Converted SysML Aeolus Simulation: Y ECI Pointing (Asset 2)

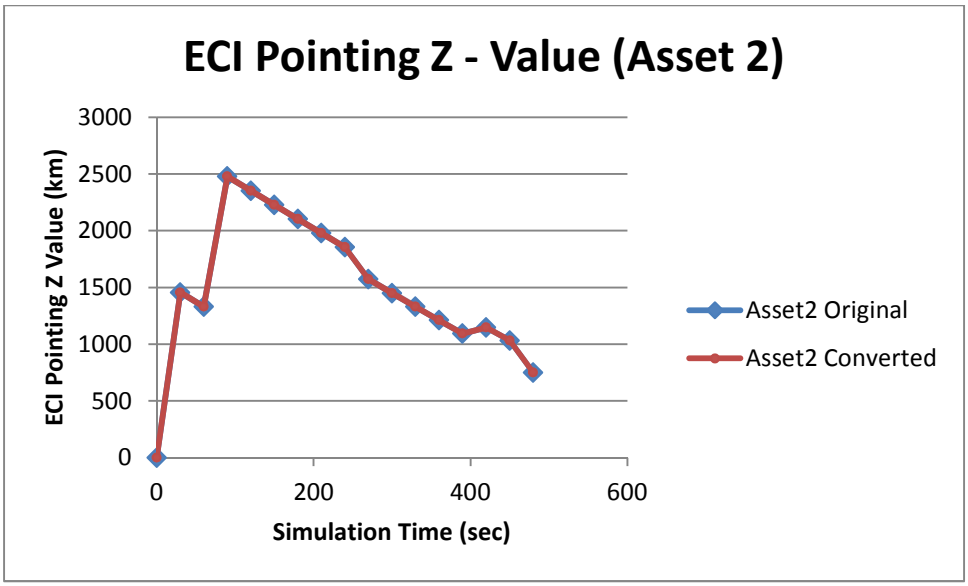

Figure 58: Original vs. Converted SysML Aeolus Simulation: Z ECI Pointing (Asset 2)

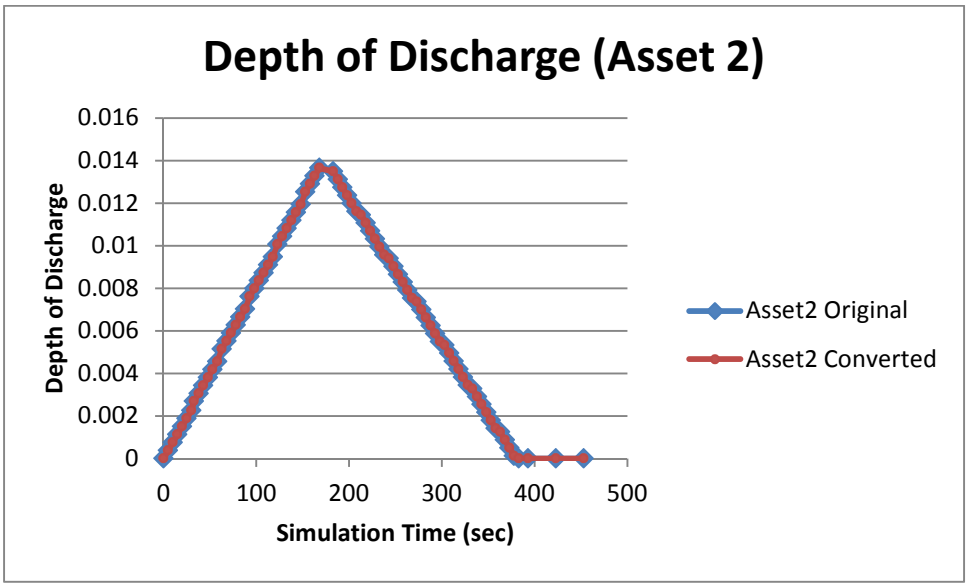

Figure 59: Original vs. Converted SysML Aeolus Simulation: Depth of Discharge (Asset 2)

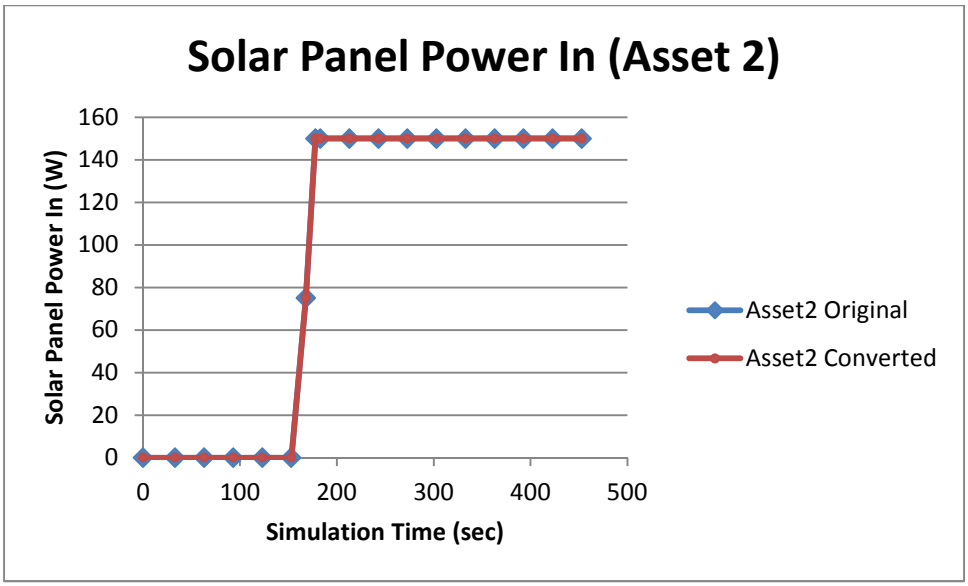

Figure 60: Original vs. Converted SysML Aeolus Simulation: Power In (Asset 2)

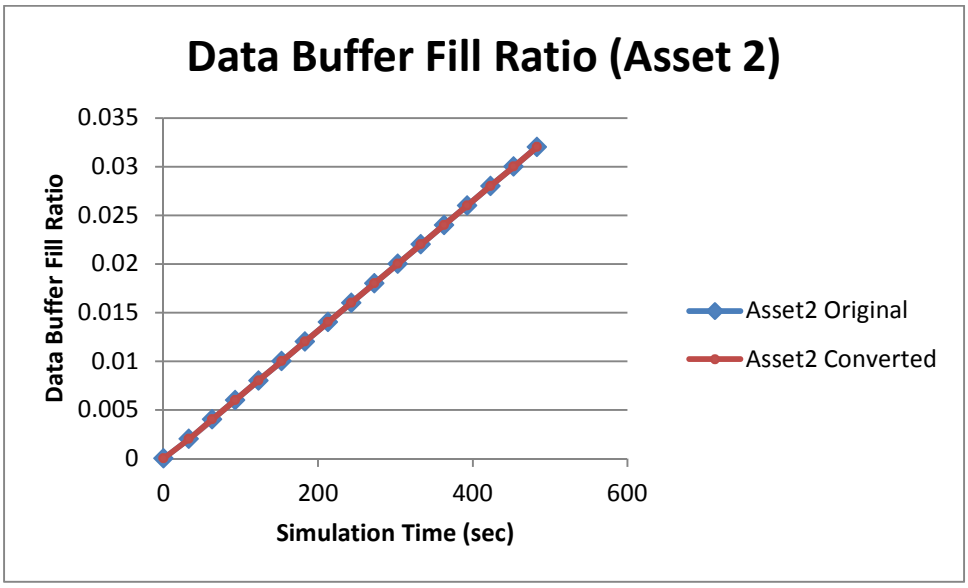

Figure 61: Original vs. Converted SysML Aeolus Simulation: Data Buffer Fill Ratio (Asset 2)

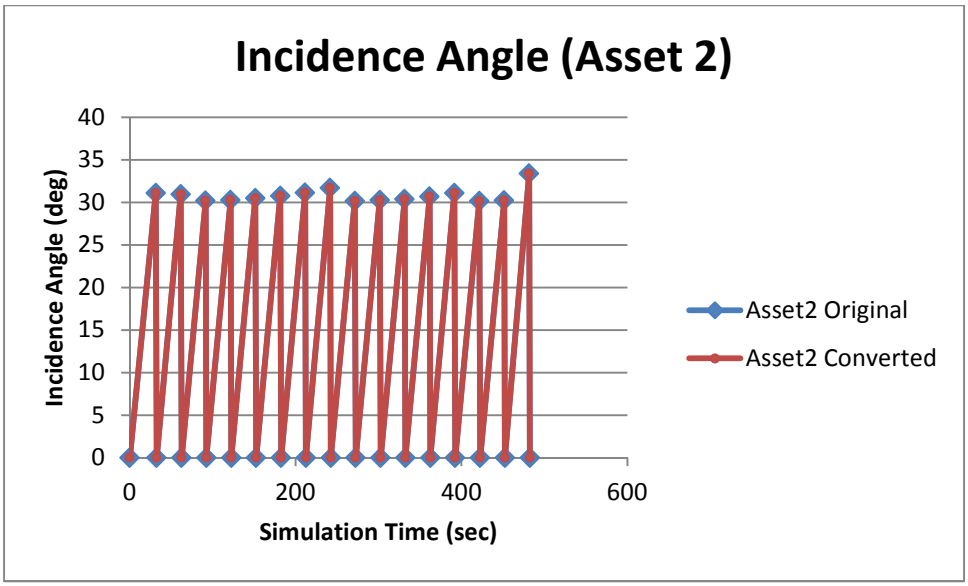

Figure 62: Original vs. Converted SysML Aeolus Simulation: Incidence Angle (Asset 2)

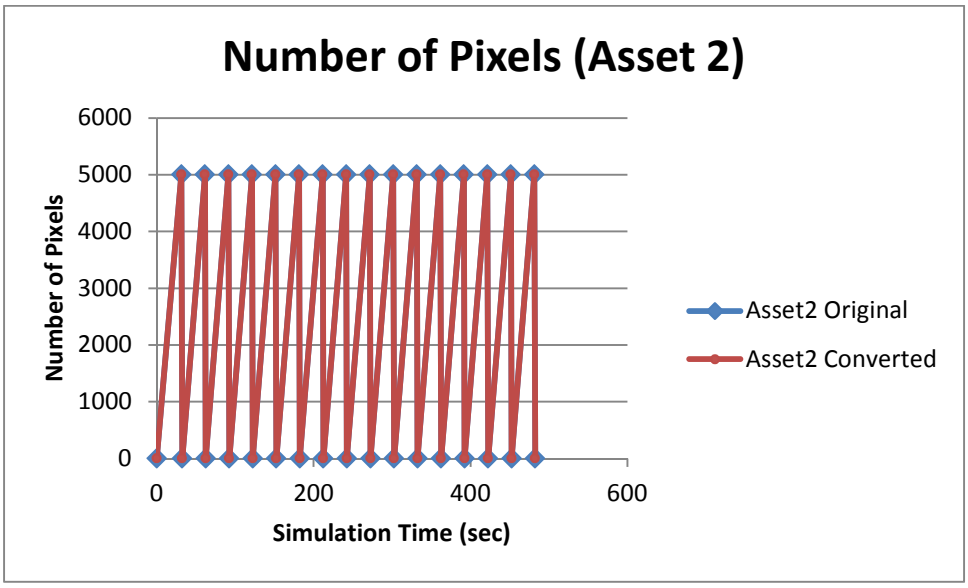

Figure 63: Original vs. Converted SysML Aeolus Simulation: Number of Pixels (Asset 2)

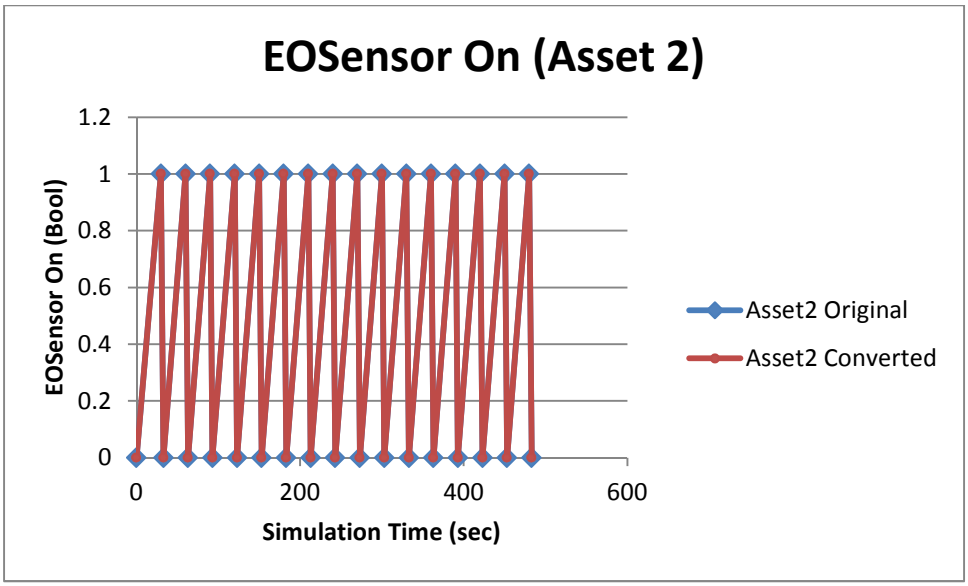

Figure 64: Original vs. Converted SysML Aeolus Simulation: EOSensor On (Asset 2)