AN ITERATION ON THE HORIZON SIMULATION FRAMEWORK TO INCLUDE
.NET AND PYTHON SCRIPTING

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
Morgan Yost
June 2016

©2016

Morgan Yost

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: An Iteration on the Horizon Simulation Framework
to include .NET and Python Scripting

AUTHOR: Morgan Yost

DATE SUBMITTED: June 2016

COMMITTEE CHAIR: Dr. Eric Mehiel, Ph.D.
Professor of Aerospace Engineering

COMMITTEE MEMBER: Dr. Kira Abercromby, Ph.D.
Associate Professor of Aerospace Engineering

COMMITTEE MEMBER: Dr. Franz Kurfess, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Dr. Ricardo Tubio Pardavia, Ph.D.
Systems Engineer

ABSTRACT

An Iteration on the Horizon Simulation Framework to include .NET and Python
Scripting

Morgan Yost

Modeling and Simulation is a crucial element of the aerospace engineering design process because it allows designers to thoroughly test their solution before investing in the resources to create it. The Horizon Simulation Framework (HSF) v3.0 is an aerospace modeling and simulation tool that allows the user to verify system level requirements in the early phases of the design process. A low fidelity model of the system that is created by the user is exhaustively tested within the built-in Day-in-the-Life simulator to provide useful information in the form of failed requirements, system bottle necks and leverage points, and potential schedules of operations. The model can be stood up quickly with Extended Markup Language (XML) input files or can be customly created with Python Scripts that interact with the framework at runtime. The goal of the work presented in this thesis is to progress HSF from v2.3 to v3.0 in order to take advantage of current software development technologies. This includes converting the codebase from C++ and Lua scripting to C# and Python Scripting. The particulars of the considerations, benefits, and implementation of the new framework are discussed in detail. The simulation data and performance run time of the new framework were compared to that of the old framework. The new framework was found to produce similar data outputs with a faster run time.

ACKNOWLEDGMENTS

Thanks to my dog for keeping me active, my family for keeping me provided for, wine and mac'n cheese for keeping me fed (and hydrated) and my best friend for keeping me social these past 5 years.

Thank you John Phillips, Rob Somerville and Clint Parker for teaching me C# and providing guidance.

A special thanks to Eric Mehiel for being a mentor and a role model in my studies, thesis, and life.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF ABBREVIATIONS	x
CHAPTER	
1 Horizon Simulation Framework Background	1
1.1 The System Engineering Design Process	1
1.1.1 Model Based Systems Engineering	2
1.1.2 Model Driven Development	3
1.2 Relevant Software Engineering Topics	4
1.2.1 The Reusable Flight Software Model	4
1.2.2 Searching the Solution Space	4
1.2.3 Dynamic Programming	5
1.2.4 Software Architecture	6
1.2.5 C# as a First Programming Language	7
1.3 Related Products	8
1.4 Thesis Statement	10
2 What is the Horizon Simulation Framework?	11
2.1 Overview	11
2.1.1 Explanation By Example– The Test Case Aeolus	11
2.1.2 Modeling and Scheduling	15
2.1.3 Modeling	16
2.1.4 Scheduling	16
2.1.5 Detailed Class Descriptions	21
2.2 Capabilities of HSF v2.3	23
2.3 V2.3 Shortcomings	25
2.3.1 Coding Style Standard	25
2.3.2 Uniform Software Architecture	25
2.3.3 Subsystem and Dependencies	26
2.3.4 Asset Schedule	29
2.3.5 Scripting	30
3 Improving the Horizon Simulation Framework	33
3.1 Motivation	33
3.1.1 Integrated Development Environment	33
3.2 Reverse Engineering from C++	36
3.3 Utilizing C# and .NET	36
3.3.1 Iterators versus IEnumerableables	36
3.3.2 Pass By Reference	37
3.3.3 Object	38
3.3.4 Searching with LINQ	38
3.3.5 Nuget Packages	38
3.3.6 Built-in XML	39
3.4 Architecture Changes	39
3.4.1 Subsystem, Dependency, Subsystem Node, Node Dependencies	40
3.4.2 State History, System State and Event	43
3.5 Class Factories	44

3.6	Incorporating Python	45
3.6.1	Choosing IronPython	45
3.6.2	Communication Between C# and IronPython	46
3.6.3	Inheriting a C# Object	46
3.6.4	Strongly Typed versus Loosely Typed	47
3.6.5	Pass By Reference versus Pass By Value	47
3.6.6	Runtime Errors	47
3.7	Uniform Coding Style	48
3.8	Unit Testing	48
4	Simulation Results	50
4.1	The Test Cases	50
4.2	C++ Baseline Results	50
4.3	C# Results	52
5	Future Work and Conclusion	58
5.1	Expanded Schedule Diagnostic Capabilities	58
5.2	Expanded Universe and Subsystem Models	58
5.3	Multi-Threading	59
5.4	Graphical User Interface	59
5.5	Agile Development	59
5.6	GitHub Quality Control	59
5.7	Model Driven Development	60
5.8	Conclusion	60
	BIBLIOGRAPHY	62
	APPENDIX: GLOSSARY	64

LIST OF TABLES

Table	Page
2.1 The Subsystems of the Aeolus Test Case	12
2.2 Modeling Elements	16
2.3 Scheduling Elements	17
2.4 Subsystem Methods	22
3.1 Subsystem Default Functionality	43

LIST OF FIGURES

Figure	Page
1.1 Illustration of Breadth-first Search from <i>Artificial Intelligence: A Modern Approach</i> (Russell and Norvig, 2010)	5
1.2 Table from Shoaib et al., 2014 displaying the scores of some commonly used programming languages for their suitability as a first programming language	8
2.1 The Subsystems of the Horizon Simulation Framework	12
2.2 Aelous Ground Targets (O'Connor, Mehiel, and Butler, 2008)	13
2.3 The Aeolus Test Case: Ground Target Captures (O'Connor, Mehiel, and Butler, 2008)	14
2.4 Horizon Simulation Framework Components (O'Connor, Mehiel, and Butler, 2008)	15
2.5 Building Schedules by Adding New Events and Preserving Old Schedules	18
2.6 Building Schedules with a Safe Copy	20
2.7 HSF GUI	24
2.8 HSF v2.3 File Tree (Class Dependencies not Included)	26
2.9 Asset Schedule in HSF v2.3 (O'Connor, Mehiel, and Butler, 2008)	30
2.10 Most Common Programming Languages Cass, 2014	31
3.1 Microsoft Visual Studio IntelliSense	34
3.2 The New HSF Architecture	39
3.3 HSF v2.3 Dependency Maps	40
3.4 HSF v3.0 Dependency Dictionary	40
3.5 State History Class that Replaces the Asset Schedule	44
3.6 Class Factory Function MSDN, 2016	45
3.7 IronPython Code to Add References to .NET and HSF libraries	46
3.8 Visual Studio Test Explorer	49
4.1 HSF v2.3 Targets Hit	50
4.2 HSF v2.3 Payload Data Subsystem State Data	51
4.3 HSF v2.3 Power Subsystem State Data	51
4.4 Targets Hit in HSF v3.0 Optimal Schedule	52
4.5 Data Buffer Fill Ratio Considering Number of Pixels in and Data Rate Out for Asset 1	53
4.6 Data Buffer Fill Ratio Considering Number of Pixels in and Data Rate Out for Asset 2	54
4.7 Battery Depth Of Discharge Considering Solar Power In and EO Sensor Power State for Asset 1	55
4.8 Battery Depth Of Discharge Considering Solar Power In and EO Sensor Power State for Asset 2	56
4.9 HSF v3.0 Run Times by Test Case	57

LIST OF ABBREVIATIONS

ADC	Attitude, Determination and Control
CLR	Common Language Runtime
CSP	Constraint Satisfaction Problem
DLL	Dynamic-Linked Library
BDESA	Big Dumb Exhaustive Search Algorithm
BFS	Breadth-First Search
CONOPS	CONcept of OPerations
DFS	Depth-First Search
DITL	Day in the Life
DP	Dynamic Programming
GUI	Graphical User Interface
HSF	Horizon Simulation Framework
LINQ	Language-INtegrated Query
MDD	Model Driven Development
MDSE	Model Based Systems Engineering
MSDN	Microsoft Development Network
SQL	Structured Query Language
STL	Standard Template Library
XML	EXtensible Markup Language

Chapter 1

HORIZON SIMULATION FRAMEWORK BACKGROUND

1.1 The System Engineering Design Process

The high level systems engineering design process is defined by the IEEE International Standard for *Systems and Software Engineering System Life Cycle Processes* (15288-2008, 2008). The technical process is composed 11 sub-processes. These processes are named in the list below.

1. Stakeholder Requirements Definition
2. Requirements Analysis
3. Architectural Design
4. Implementation
5. Integration
6. Verification
7. Transition
8. Validation
9. Operation
10. Maintenance
11. Disposal

In an ideal engineering process, these 11 steps would be performed in an iterative cycle that improves upon the previous version. The first 2 processes align with the preliminary design phase of the overall system development process. This phase is referred to as Phase A in the classical aerospace design textbook, *Space Mission Analysis and Design*. Phase A is when a concept of operations (CONOPS) that satisfies mission requirements is developed. Validating that mission requirements are met with the CONOPS is an important step before the rest of the aerospace design phases, B-E, can be completed. As a result, this verification process is a crucial step in the aerospace design process, and an error in the CONOPS may result in wasted project budgets, slipping schedules, or worse, mission failure.

Modeling and Simulation In some engineering disciplines, verifying a design can be done by simply implementing and testing the solution. In the aerospace industry where systems can cost millions of dollars and mission can cost billions, building and testing an entire mission design would be impractical. As a result of a need for a cost effective method to implement and verify a system, system models and a simulations are traditionally utilized. Verifying a model of the CONOPS in a simulation cements the CONOPS as a high level solution to the mission requirements, and allows the project to progress to the next phase where the technical solution is baselined.

Modeling and simulation takes on many forms such as scaled prototypes, mathematical equations and Computer Aided Drafting (CAD). What all of these methods have in common is that they allow the user to create a custom model of their design to be tested to some extent. According to Loper, a "model is a physical, mathematical, or otherwise logical representation of a system" and a "simulation is a method for implementing a model over time" (Loper, 2015) By creating smaller scale or computer based models of a potentially costly system, the technical life cycle can be executed for a representative system at reduced overall cost. In the specific case of space mission design, modeling and simulation is especially important because of the high cost of space missions, the relatively long timeline of the project and the inaccessibility of the operational environment.

1.1.1 Model Based Systems Engineering

The process of defining CONOPS is typically executed by creating documentation. If the CONOPS are drawn up in a report, then for the requirements to be analyzed and the architecture designed, the report must be interpreted by the stakeholders and engineers (Micouin, 2014). In her book, *Model Based Systems Engineering: Fundamentals and Methods*, Micouin elaborates on how this interpretation of the documentation is not objective, and therefore error prone. Instead, she advocates for the use of models to define these requirements because when the requirements are "objectified, the specifications then become exactifiable; in other words, the stakeholders are put into a situation where they are able to decide on the validity of behaviors observed, just as they could do on a test bench" (Micouin, 2014). The use of models to define requirements is known as Model Based Systems Engineering (MBSE).

1.1.2 Model Driven Development

While MBSE helps mitigate the misinterpretation of the CONOPS as they are transitioned into the baseline technical specification of Phase B, the efficiency of this transition can be further improved by Model Driven Development (MDD). The concept of MDD states that, since the model is already made for the purpose of requirements analysis, it should be reused in subsequent phases to eliminate repetitive work. As applied to the phases of aerospace design, the model, or CONOPS, that is created in Phase A could be automatically translated to the technical baseline, in Phase B. The automatic generation of the technical baseline from models has many benefits for the development process and the quality of the product. If the specification is created automatically from the model, the development time can be shortened because time is spent developing only the model rather than creating a model then a specification. Also, changes and improvements that are made to the model during the iterative engineering process are automatically reflected in the specification and don't need to be manually added. Since the model is also used in the simulation that verifies its functionality, and the specification is made directly from the model, verification only needs to occur once rather than twice. The consistency of the automatic technical baseline generation may also mitigate human error in conversion from a model to the specification, thus improving the quality of the end product.

Flight software is a good example of an element of the aerospace system that can truly follow MDD by utilizing the model from inception to fabrication. "A key premise behind MDD is that [software] programs are automatically generated from their corresponding models" (Selic, 2003). MATLAB's Simulink™ is one of many programs that provides this MDD service. With Simulink, a designer can create a model and run it through simulations to verify its functionality. Once the designer is satisfied with the system's performance, he can have the model autocoded into software written in the C programming language to run on the actual physical system. In this way, modeling and simulation can be pervasive throughout the phases of the aerospace design process.

1.2 Relevant Software Engineering Topics

For the unfamiliar reader, the glossary provides definitions of terms used in this document specific to software engineering and may be helpful as a quick reference while reading this section. It is assumed that basic programming concepts and terminology including object oriented programming are familiar to the reader.

1.2.1 The Reusable Flight Software Model

The fundamental ideas of reusable flight software are especially applicable to the concepts that helped develop the architecture of the Horizon Simulation Framework. Hawthorn, Weber, and Scholten identify ways to create reusable, modular and scalable flight software for aerospace applications. The author argues that all flight software systems can be created modularly if the system is decomposed into a semantics catalog– "a hierarchical classification of related concepts that present a logical view of the problem space" (Hawthorn, Weber, and Scholten, 2014). By discerning subsystems for their related concepts, a designer can achieve plug and play functionality for their systems with the ability to exchange subsystems as requirements change between missions. The ability to swap out subsystems, of course, is made much more possible by the implementation of standard interfaces and a standard method to communicate data, or "ontology" (Hawthorn, Weber, and Scholten, 2014). The authors also advise that rather than have mission specific parameters hard-coded within the software files, create a database of mission specific variables that can be queried for use in general equations. A good example of this is the gains for a proportional derivative (PD) control law, or the transformation matrix for the mounting location of a star tracker.

1.2.2 Searching the Solution Space

The classical textbook on artificial intelligence, *Artificial Intelligence: A Modern Approach* (Russell and Norvig, 2010), defines a problem in general as being comprised of 5 elements. The initial state, actions that can be performed, the transition model, the goal test and the path cost. If a problem is posed to a problem solving algorithm, a solution, or sequence of actions to arrive at a goal, can be found if it exists. Typically, the problem solving algorithm is simply a search algorithm that searches the solution space for the desired goal. In cases when all possible solutions wish to be known, an uninformed

search strategy can be used. One of the many potential uniformed search strategies is breadth-first search (BFS). In breadth-first search, the shallowest unexpanded node is chosen for expansion– or to be explored– first (Russell and Norvig, 2010). Figure 1.1 illustrates the process of breadth first search.

FIGURE 1.1: Illustration of Breadth-first Search from *Artificial Intelligence: A Modern Approach* (Russell and Norvig, 2010)

Once all the nodes have been expanded, the entire solution space has been explored and all possible solutions that exist have been found. One requirement of BFS, however, is that the solution space is discrete, as continuous spaces result in an infinite branching factor (Russell and Norvig, 2010). Another consideration when using BFS is that it requires b^{d+1} amount of memory where b is the branching factor and d is the depth of the tree (Russell and Norvig, 2010). As a point of comparison, depth-first search (DFS) expands the deepest node first to follow a path all the way to its end before restarting at the first node to follow a new path. DFS only needs to store the current path being expanded and all previous successful paths, whereas BFS must store all nodes because the solutions are not apparent until all nodes have been evaluated. However, as an application in Dynamic Programming (1.2.3), breadth-first search proves to be a good choice for the Horizon Simulation Framework as discussed in 2.1.4.

1.2.3 Dynamic Programming

Dynamic Programming (DP) was coined by the applied mathematician, Richard Bellman. Lew and Holger elegantly phrases its definition as: "Dynamic programming is a method that in general solves optimization problems that involve making a sequence of decisions by determining, for each decision, subproblems that can be solved in like fashion, such that an optimal solution of the original problem can be found from optimal solutions of subproblems." So while both DP and BFS explore the entire solution space, DP searches for the optimal solution by assigning a value to the solution found

by each subproblem. Bellman's principle of optimality further elaborates on the ability to find an optimal policy, also known as path or schedule.

Principle of Optimality: An optimal policy has the property that whatever the initial state and initial conditions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions. (Bellman, 1954)

Moghaddam and Usher discuss an application of dynamic programming in "Preventive maintenance and replacement scheduling for repairable and maintainable systems using dynamic programming" (Moghaddam and Usher, 2011). Similar to the issue of memory consumption with BFS, the authors note that "one of the well-known difficulties associated with applying dynamic programming to real world problems is the so-called 'curse of dimensionality' due to increasing number of state variables in each stage of recursion." In order to mitigate for this problem, the authors find success in using a method introduced in "Branch-And-Bound Strategies for Dynamic Programming". The branch-and-bound strategy incorporates trimming the graph before every new node is explored. This helps prevent unnecessary computation in exploring paths that already show less promise than others as well as prevents expending valuable memory resources. When combining the concepts of dynamic programming and branch-and-bound in layman's terms, it can be said that as each subproblem is solved, the non-optimal solutions can be eliminated in favor of the optimal. While this method obviously does not find all possible solutions because some paths are discarded along the way, it finds as many solutions that are within the "bound", which can be sufficient for the developer that has a general idea of what he is looking for.

1.2.4 Software Architecture

One major decision when designing a software product is the architecture of the code-base. Decomposing the functionality of the code into modules is essential for teams to be able to all work on the same code base and for the maintainability of the code. *Software Architecture in Practice* (Bass, Clements, and Kazman, 2003) discusses some guidelines to follow when architecting the structure of a software project. These guidelines are considered to be good architecture practices by the authors and are paraphrased below.

1. The architecture should feature well-defined modules that are responsible for information hiding and separation of concerns.
2. Each module should have a well-defined interface that encapsulates or "hides" changeable aspects (such as implementation strategies and data structure choices) to allow their respective development teams to work largely independently of each other.
3. The architecture should only depend on a commercial product if structured such that changing to a different product is straightforward and inexpensive.
4. Modules that produce data should be separate from modules that consume data so that if new data is added, the separation of control allows for a staged (incremental) upgrade.
5. The architecture should feature a small number of simple interaction patterns—the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.

A key takeaway from these guidelines is that responsibility should be dispersed throughout multiple modules with well defined interfaces so that over time, portions of the software can be replaced or upgraded and developers can even be changed.

1.2.5 C# as a First Programming Language

Another important decision in software design is choosing a language to write the software in. In many cases, this decision is based on the software requirements and a language is selected that will satisfy these requirements most efficiently. However, this thesis utilizes a different criteria that aligns with its objectives that will be discussed in greater detail in Chapter 3. The ability to learn the programming language, especially as a first programming language is of special interest. Characterizing the learnability of a programming language is not easy to do as much of the applicable metrics are subjective and according to preference. Shoaib et al. uses a weighted combination of technical and environmental features to determine a suitable language for a first programming language (FPL). Figure 1.2.5 shows the scores calculated by the authors, where a higher score indicates better suitability as a first programming language. It should be noted that Java, Python and C# rank the highest in this study's objective analysis.

Table 25. Overall score for widely used programming languages (sorted based on score with customized weights).

Scores with higher weightage of "Demand in Industry" and "Easy Transition"		
Languages	Unbounded L_s	Bounded L_s'
Java	12.18	0.76
Python	9.92	0.62
C#	9.15	0.57
Ada	8.70	0.54
C++	7.96	0.50
Modula-2	7.83	0.49
Pascal	6.89	0.43
C	5.20	0.32
Fortran	4.80	0.30

doi:10.1371/journal.pone.0088941.t025

FIGURE 1.2: Table from Shoaib et al., 2014 displaying the scores of some commonly used programming languages for their suitability as a first programming language

1.3 Related Products

As previously mentioned, MATLAB Simulink™ provides an environment to model and simulate space missions, but this is only one of a myriad of products that seek to provide this functionality to the user. The advantages and shortcomings of a few of these products are described below.

MATLAB Simulink™ The modeling features and libraries of MATLAB and Simulink are well respected in the aerospace community as exceptional tools to verify subsystem performance and functionality. The large selection of pre-made modeling components makes creating a model fast and easy. Simulations can be run by feeding data into the system and analyzing the system's response. In order to create schedules, the user can purchase an add on product called SimEvent. "SimEvents® provides a discrete-event simulation engine and component library for analyzing event-driven system models and optimizing performance characteristics such as latency, throughput, and packet loss" (MathWorks, 2016). The MATLAB with Simulink and SimEvents license costs almost \$200 for students and \$8,000 for the standard license. While SimEvents allows the user to create schedules for the model to run through, the creation of exhaustive schedules would be tedious, and thus the user is missing out on the opportunity to learn from an extensive set of failed and successful scenarios. Other software programs

that have comparable capabilities to Simulink are MapleSim, Wolfram System Modeler and SimulationX.

Questa® Formal Verification Mentor Graphics has developed a tool that tests models for correctness. "Questa Formal Verification analyzes the behavior of the design to identify all design states that are reachable from the initial state. This analysis allows Questa Formal Verification to explore the whole state space in a breadth-first manner, in contrast to the depth-first approach used in simulation" (*Questa Formal Verification*). Questa's wide range of verification applications make it a viable option for analyzing the simulation solution space, however, it does not support the modeling segment to the extent that Simulink does, and it is unclear how much support it provides for aerospace subsystems. The Questa Formal Verification platform seems more targeted for low-level requirements analysis where a detailed model is provided to the system. Mentor graphics provides Questa Formal Verification as well as supporting applications for a price.

European Space Agency Virtual Spacecraft Design The European Space Agency (ESA) has dedicated efforts to creating a Virtual Spacecraft Design (VSD) software program that allows users to drag and drop actual subsystem models into a graphical user interface. By connecting existing models of subsystems, a model of the a unique system can be synthesized. The Virtual Spacecraft Design software focuses on the modeling element of modeling and simulation by providing extensive functionality to create the model for the purpose of defining the system. Specifications that can be defined in the model include "requirements, functional architecture, physical architecture, Spacecraft operation" (ESA, 2016) and verification and fabrication plans. VSD provides impressive modeling capabilities that support the use of MBSE, but it is more focused on the technical baseline phase of the spacecraft design process than the initial proof of concept phase. For that reason, it requires the designer to have a precise idea of the design and does not support exhaustive scenario simulations.

1.4 Thesis Statement

The Horizon Simulation Framework seeks to combine the topics discussed in this chapter to solve the design verification problem. The framework is in an iterative cycle of improvement to continue to make it a useful and contemporary modeling and simulation tool. The purpose of this thesis is to present the newest iteration of Horizon Simulation Framework, v3.0. The updated framework takes advantage of current software development technologies such as C# and Python.

The following chapters of this thesis will go on to explain the goals for the next iteration of the framework and how these goals were achieved. Chapter 2 defines the Horizon Simulation Framework (HSF) and elaborates on the shortcomings of the previous version, v2.3. Chapter 3 addresses solutions to these shortcomings and offers insight on how to improve the framework in the newest version, v3.0. Chapter 4 provides the simulation results that verify that the new framework is operational as well as the performance results of simulation run time. Chapter 5 discusses opportunity for future work and concludes the thesis.

Chapter 2

WHAT IS THE HORIZON SIMULATION FRAMEWORK?

2.1 Overview

The Horizon Simulation Framework (HSF) is a modeling and simulation framework for verification of system level requirements. HSF allows users to create a low fidelity model of their system by providing the framework with rough specifications of subsystems and their interactions or dependencies with one another. The user also specifies the parameters of the simulation such as initial conditions, time step size and duration that is then used to create a Day-in-the-Life (DITL) like scenario to test the model. As the DITL simulation is run, the user learns about successful system use cases as well as system failure points. While the success of the system is important, information learned from bottlenecks and missed requirements is potentially more valuable to the iterative engineering design process. The DITL simulation seeks to be exhaustive so that all possible scenarios that can be executed by the system are tested in order to accumulate a large amount of data for post processing and analysis.

2.1.1 Explanation By Example– The Test Case Aeolus

For validation purposes, an imagined mission named Aeolus is used to test the framework. Aeolus is a two satellite constellation whose mission is to image as many ground targets as possible. Each satellite is composed of the subsystems described in Table 2.1, and their dependencies are shown in Figure 2.1. Aeolus can be used to expand on the purpose of HSF and as a demonstration of the framework's capabilities. Previous versions of the framework created and utilized the Aeolus test case, which, for consistency sake, will also be used in this thesis.

TABLE 2.1: The Subsystems of the Aeolus Test Case

ADCS	The Attitude, Determination and Control Subsystem is responsible for ensuring that the satellite is pointed to the target while the satellite has line of sight to the target and an imaging task is scheduled. The default ADCS specifications require that the system have 10 seconds to slew before the system can capture an image with the EO Sensor.
EO Sensor	The Earth Observing Sensor takes images of a target when an imaging task is scheduled. The EO sensor can take images with high, medium or low resolution. The default values are 15000, 10000, 5000 pixels respectively.
SSDR	The Solid State Data Recorder saves data from the EO sensor until a downlink task is scheduled. The SSDR has a default storage capacity of 5KB.
COMM	The Communications subsystem relays data to the ground station when a ground station has line of site and a downlink task is added.
Power	The power subsystem maintains the depth of discharge (DOD) of the system's battery by taking into consideration the power consumed by the other subsystems and power generated by the solar panels. The battery had a default capacity of 1000 kW-hr.

FIGURE 2.1: The Subsystems of the Horizon Simulation Framework

The Aeolus mission requires the constellation to image the 296 ground targets shown on the map in Figure 2.2. It is the task of the user to specify how many times each target

may be imaged, which are the most important and how long the system has to complete its mission. These values are used to create the heuristic that is used to assign values to schedules in the schedule evaluator.

FIGURE 2.2: Aelous Ground Targets (O'Connor, Mehie, and Butler, 2008)

A major goal of HSF is to provide plug-and-play functionality by designing the framework with modularity and flexibility in mind. In order to provide this modularity without exposing the source code to the user and forcing them to recompile the program every time a modification is made, the HSF executable program has the ability to interface with externally developed files and scripts. These external files have the ability to modify the system model structure and functionality without modifying the framework itself.

To enable quick set up, basic subsystems are all built into the framework. However, Extended Markup Language (XML) input files provide the user with the ability to modify exposed attributes of the subsystems, dependencies, constraints, ground target locations, simulation parameters, equations of motion and the way schedules are valued. For instance, if a spacecraft developer had the Interface Control Document (ICD) for the Solid State Data Recorder (SSDR) subsystem that he had in mind for the mission, he would be able to specify the memory capacity of the HSF model to more accurately model the data flow within his system. The user can also overwrite functionality included in HSF and provide a subsystem not included in the executable with the framework's feature to allow scripted classes to interact with the executable. This

provides the user with the ability to define a model from scratch by simply redefining attributes and functions.

In order to have a simulation environment to test the model in, HSF creates all possible scenarios, or combinations of events, to image the ground targets in all possible orders. With each DITL scenarios, the model is advanced through the scenario to see if its state can progress from one time step to the next without violating constraints or the system's capabilities. After all the scenarios are executed, the optimal schedule as well as diagnostic information for the failed schedules is returned to the user for analysis. A sample of the schedule output was found by O'Connor, Mehiel, and Butler and is pictured in Figure 2.3.

FIGURE 2.3: The Aeolus Test Case: Ground Target Captures (O'Connor, Mehiel, and Butler, 2008)

O'Connor, Mehiel, and Butler elegantly explain the importance of diagnostic information and how it can be used to improve a system via the identification of bottlenecks and leverage points. "Here, a bottleneck within the system is any system design feature or parameter that is a hindrance to the overall utility of the system" (O'Connor, Mehiel, and Butler, 2008). An example of this in the Aeolus test case could be imagined in the scenario when power subsystem battery level is too low, or the ADC subsystem doesn't have enough time to slew to a target. "Leverage Points within the system are system design elements or parameters that if changed only slightly result in a proportionally large benefit to the system utility" (O'Connor, Mehiel, and Butler, 2008). When this

state data is reported to the designer, the designer has the ability to make an informed decision to perhaps switch to a larger battery or higher torque wheels in order to improve the capabilities of his system to satisfy design requirements. The designer also now has a better idea of the performance to expect from his system in the operational phase of the program as well as a estimate for an optimized schedule of operation.

2.1.2 Modeling and Scheduling

The original HSF framework was designed to provide the user with an adequate amount of customization capabilities while also keeping in mind that the framework should perform as much work for the user as possible. As stated by O'Connor, Mehiel, and Butler, "the design of the framework is based on the application of modularity of system modeling and scheduling, flexibility with respect to the fidelity of the simulated system, and utility of the supporting libraries in the framework." As such, the software was divided into a modeling segment and a scheduling segment with the idea that the user would have the freedom to customize the model in the modeling segment and the framework would know how to ingest this information and execute the scheduling and evaluation of the system. Figure 2.4 presents the interactions between the two segments with the modeling segment, or system, in the yellow box and the scheduler, or simulation, in the blue square.

FIGURE 2.4: Horizon Simulation Framework Components (O'Connor, Mehiel, and Butler, 2008)

2.1.3 Modeling

The modeling aspect of HSF defines how the system consumes the schedule. The user must specify the subsystems, dependencies and constraints that make up the system. These terms, as well as other terms relevant to the modeling elements within HSF are defined in Table 2.2.

TABLE 2.2: Modeling Elements

Dependency	A dependency defines a relationship between two subsystems. The purpose of the dependency is to allow subsystems to relay necessary information without requiring them to know the inner workings of each other. This allows the subsystems to be truly modular by limiting communication through a pre-specified interface. A dependency function is the relay of information from one subsystem to another. A dependency collector is a function that calls all dependency functions for specific subsystem and combines the data in a customizable way. It should be noted that there can be no circular dependencies in the model.
Constraint	A constraint can be applied to a particular state of a subsystem to indicate that a value is constrained to a specific range of values or options. This allows the user to require a subsystem to operate within the capability of the system. In order to ensure the system fails fast, the subsystems with the constraints are evaluated first.
Subsystem	The subsystem is the fundamental modeling element of HSF. By specifying a subsystem, the user specifies the capabilities of the system and how the system's state variable changes over time. The subsystems are passed tasks and must determine if they can perform them as well as update their state to indicate the task being performed.
Asset	An asset is a simple container that unites subsystems that share the same dynamic state such as position and velocity. The asset is also used to maintain the state of the subsystems grouped in the same asset as well as task them together so that each asset could be performing a different task.
System	The culminating element of the modeling components is the system. The system is a combination of subsystems that are related through dependencies and verified by their constraints. The system also relays the state, position and environment information to the subsystems.

2.1.4 Scheduling

Scheduling in HSF is the operation that creates DITL scenarios for the modeled system to execute. Scheduling entails combining events to create a chronological list of

events that the system then performs. In order to discuss how the Horizon Simulation Framework develops schedules, it is important to first define the ontology use for scheduling in HSF. Table 2.3 outlines the key scheduling elements of the Horizon Simulation Framework.

TABLE 2.3: Scheduling Elements

Dynamic State	The dynamic state is an element specialized for maintaining information about the dynamics, such as position and velocity, of the system.
Task	A task is the goal of an asset at each simulation time step. It consists of a target and the type of action to be performed at the target.
State	The state is a system level catalog of the current state of the subsystems (i.e. battery depth of discharge or memory usage). Each subsystem has a key that allows them to access their state information from the high level catalog. The time history of the state throughout the simulation time is maintained in a chronological collection of states.
Event	An event is the combination of a task and state. For every simulation time step there is one event and a task may be scheduled at any point within an event.
Schedule	A schedule is simply a chronological collection of events. The tasks within every event are proposed to the system and only the schedules that can be performed are output by the simulation.

The Algorithm The HSF scheduling algorithm is the backbone to the framework. In order to ensure completeness, a breadth-first search of the solution space is used. This recursive algorithm generates all possible combinations of events so that the system can be thoroughly tested for all possible schedules. The event combinations are made keeping in mind that "no action" is also a possible event. This is accomplished by tacking on new events one at a time, and preserving old schedules without the new event to simulate the "no action" event. This algorithm is illustrated through example in Figure 2.5. The time increments used by the scheduler are provided by the user as the simulation time step, thus discretizing the simulation.

FIGURE 2.5: Building Schedules by Adding New Events and Preserving Old Schedules

Due to the high space complexity of breadth-first search, dynamic programming concepts introduced in Section 1.2.3 are applied. As a result, Algorithm 1 is developed with the use of heuristics and branch-and-bound dynamic programming to be memory efficient.

For each timestep, k

input : All possible schedules from t_0 to t_k

output: All possible schedules from t_0 to t_{k+1}

Generate list of all possible tasks that can be performed at t_{k+1}

```

foreach schedule  $s$  of all possible schedules do
    foreach task  $t$  of all possible tasks do
         $r \leftarrow \text{SafeCopy}(s)$ ;
         $r \leftarrow \text{AddNewTask}(t)$ ;
        Add  $r$  to the list of all possible schedules
    end
end
foreach schedule  $s$  of all possible schedules do
    if  $\text{CanPerformNewTask}(s) == \text{FALSE}$  then
        |  $\text{Eliminate}(s)$ 
    end
end
foreach schedule  $s$  of all possible schedules do
     $\text{score} = \text{EvaluateWithHeuristic}(s)$ ;
    if  $\text{score} < \text{threshold}$  then
        |  $\text{Eliminate}(s)$ 
    end
end

```

Result: Only the best possible schedules remain

Algorithm 1: Horizon Simulation Framework Scheduling Algorithm

Safe Copy One very important element of the scheduler is how the schedules are built up using the safe copy. Drawing from the concept of dynamic programming, once one subproblem has been solved, it can be used within larger subproblems. Lew and Holger explain that "for DP to be computationally efficient (especially relative to evaluating all possible sequences of decisions), there should be common subproblems such that subproblems of one are subproblems of another." For this reason, the old schedule is used directly within the new schedule by simply creating a safe copy of the old schedule before the new task is added onto it. The safe copy accomplishes the following three things:

- Since the events in the previous schedule have already been evaluated, they won't need to be evaluated again.
- Since the events in the previous schedule are already in memory, they won't need to be stored again.
- Since the schedule with the new task may be manipulated or destroyed, it won't affect the schedules it is copied from.

Figure 2.6 graphically illustrates how new schedules are safely created from old ones. It can be seen that memory is saved by creating three new schedules by safely referencing the previously evaluated one.

FIGURE 2.6: Building Schedules with a Safe Copy

Schedule Elimination Another way to mitigate the fast growth of memory consumption is by limiting the number of schedules to keep in memory. This is derived from the branch-and-bound concept detailed by Morin and Marsten, 1975. In HSF, the bound is simply a maximum number of schedules— specified by the user— that HSF seeks to maintain by removing schedules as the maximum number is breached. The last loop in Algorithm 1 shows the use of heuristics to eliminate schedules that don't rank highly. This cost function, or "policy function" in Bellman's words, is created by the user and informs the framework what the user values most in the system. As discussed in Section 1.2.3, elimination of the schedules before they are completely evaluated will still allow for the best ranking schedules to be found in accordance with Bellman's Principle of Optimality and proved in Morin and Marsten, 1975. However, just as schedules must be safely copied from old ones, they must be destroyed safely as eliminating a schedule that is being referenced from a future schedule will result in a null pointer.

2.1.5 Detailed Class Descriptions

The elements described in Tables 2.2 and 2.3 provide a good understanding of the basic components within HSF. However, to gain a deeper understanding of the code base for the purpose of analysis, the software implementation of these members and the fundamental classes that support them must be discussed.

SystemState The system state is a container for all the types of data that can be held by the system. As such, it has a dictionary collection of integer, double, boolean and Matrix data that each subsystem can write to using their StateVarKey. The SystemState is in the schedule rather than the subsystem in order to allow multiple schedules to run simultaneously while only instantiating one instance of the system. In this way, the SystemState performs the duty of maintaining information while system is used solely to process tasks.

StateVarKey The dictionaries within the system state correlate state data to a subsystem through the StateVarKey. This means a subsystem knows how to find its own state data because it knows its own StateVarKey. Because the SystemState is like a bulletin board that is visible to anyone with access to the state, it is important to restrict the way the state can be modified and accessed. HSF follows the guideline that subsystems are only allowed to modify their own state and state data can only be retrieved from the state by the subsystem that the state data belongs to. The StateVarKey is a templated class so that the type of the StateVarKey is an indication to the SystemState of which dictionary (integer, boolean, etc) to look up the data in. This use of a templated class allows for a single function to be overloaded by parameter type rather than one function per dictionary access.

Task The Task class is responsible for correlating a Target to an action, or TaskType. A Target is a general class to hold information about anything the system may be targeting. In the case of Aeolus, the Target holds ground target information for imaging or relaying data. A Target could be anything the system has a goal to achieve. The TaskType simply instructs the system what to do at the Target such as image or downlink.

Event The Event is the fundamental scheduling element. The simulation progresses Event by Event, as a result, the length of the Event is the same as the simulation time step. Each asset can be tasked once per Event.

Subsystem The Subsystem class is an abstract class that is the fundamental modeling element of the framework. A Subsystem is required to hold its StateVarKeys, a list of its dependent subsystems, a local dictionary of its dependent subsystems' dependency functions, a name, the Asset that it belongs to and an IsEvaluated flag. The Subsystem also has private fields to hold the parameters from the current event that might be needed such as the previous and current state and the current Task. The abstract class provides basic implementations of the 3 methods described in Table 2.4 and can be overridden by the deriving subsystem.

TABLE 2.4: Subsystem Methods

CanPerform	Input: Event, Universe. Output: Bool. The CanPerform method is how the state of the system progresses from one time step to the next. With each proposed Event, if the Subsystem can perform the requested Task given the current SystemState, then it updates the SystemState to reflect the Task being performed and returns true. If something restricts it from performing a Task, it simply returns false without updating the SystemState.
CanExtend	Input: Event, Universe, double. Output: Bool. The CanExtend method is very similar to the CanPerform method except the Subsystem is not being requested to perform a Task, but rather simply extend its state to a time input as a parameter. The abstract subsystem CanExtend will simply return true if the time input as a parameter is still within the same Event.
DependencyCollector	Input: Event. Output: HSFFProfile. The DependencyCollector method of the abstract Subsystem simply calls all the dependency functions within the local dictionary of dependency functions and adds all their results.

Asset An Asset is an encapsulation of the Subsystems in order to group them by DynamicState. Since each Asset may be performing a different Task during each Event, the Asset is also the driving element for tasking the Subsystems.

System The System brings together all the information within the Assets, Subsystems, Constraints and Universe to create a complete picture of the system model.

Access Not all Tasks can be scheduled at any moment in time. Some Tasks are dependent on the availability of a state or opportunity. The Access maintains the availability of a Task for an Asset.

SystemSchedule The SystemSchedule is the ultimate element of the scheduling aspect of the framework. The system schedule holds a chronological list of Events– the building blocks of the schedule. Because a schedule can also be evaluated based on its state and the tasks it contains, the schedule contains a field for the value of the schedule for comparison purposes.

2.2 Capabilities of HSF v2.3

Beyond the basic capabilities of the scheduler and built in models of HSF, v2.3 also contains extra features to allow the user to further customize their simulation without directly editing the source code. These features are listed below in order to provide context on the state of the framework before discussing areas of improvement.

GUI Prototype There is a vision for the Horizon Simulation Framework to one day be completely GUI based so that the user need not generate the XML files or scripts required to model his system. This vision has been implemented in a prototype that allows users to drag and drop customizable subsystems then connect them to show dependencies. Figure 2.7 shows a sample of the GUI and helps elaborate on the ultimate goal of HSF.

FIGURE 2.7: HSF GUI

XML Specification The simulation parameters, scenario specification and subsystem specification (including dependencies and constraints) can all be input to the framework through XML configuration files. These files define properties of the simulation such as time step, environments, parameters of the subsystems and initial conditions. The use of the XML standard for input files allows for easy to modify template files as well as the eventual auto generation of inputs created by the GUI.

Scripting The modeling of subsystems through scripting was previously supported by the LUA language. The user was given the ability to define specific functions in LUA and HSF managed binding the LUA function to a C++ function call. The user had access to script certain functions of the subsystem, dependencies and constraints. The reference to these scripted functions would be relayed through the Model XML file so that the framework could bind it appropriately.

Multi-Threading HSF v2.3 supports multi-threading via the multi-threaded scheduler. This scheduler can process multiple different schedules simultaneously on separate threads in order to accelerate the schedule validation process. This feature must be kept in mind in all future iterations of HSF as the architecture, design and implementation must be thread safe.

Genetic Scheduling Algorithm As a substitute to the BFS scheduling algorithm, a genetic algorithm scheduler was developed as an option for users. This algorithm will not be included in HSF v3.0 because it is not part of the baseline Aeolus test case, but should be included in future iterations of the framework.

2.3 V2.3 Shortcomings

While the 2.3 iteration of HSF is functional and useful, there are improvements that can be made in order to create a more productive work environment for developers, as well as an easy to use software product for users. The following sections highlight issues that were uncovered during the reverse engineering process from C++, as well as improvements that weren't possible in C++, but are in C#.

2.3.1 Coding Style Standard

With multiple contributors throughout the iterations on the framework, there is no consistent coding style to the code base. While this may seem like a matter of preference, consistent coding standards make it easier for other developers to understand code they didn't write. "Coding standards provide a common base for exchanging and understanding work outputs of individual developers" (Maruping, Zhang, and Venkatesh, 2009). Section 1.2.4 discusses guidelines for good software architecture, and not surprisingly, coding style is an important element. Bass, Clements, and Kazman advise that "the system should do the same things in the same way throughout. This will aid in understandability [and] reduce development time."

2.3.2 Uniform Software Architecture

After the original conception of HSF, the focus of the more recent iterations has been on adding features and improving functionality. With each progression, the framework has become more capable, but the architecture was overlooked and did not iterate along with the code. As a result, the new modules were tacked on to the existing code base without much consideration for the guidelines outlined by Bass, Clements, and Kazman. This resulted in confusing file trees as seen in 2.8, and the dependencies between these files were even more complex. While the many different namespaces in the C++ code base resulted in a confusing structure to navigate through and adhere to,

2.3.3 Subsystem and Dependencies

FIGURE 2.8: HSF v2.3 File Tree (Class Dependencies not Included)

attitude, determination and control (ADC) subsystem might perform calculations in revolutions per minute. This allows all subsystems to operate in accordance with the specification document they may be derived from. However, in a typical spacecraft there are dependencies between subsystems. For instance, the power subsystem would be affected by the ADC subsystem commanding its actuators and consuming power. In modeling and simulation, this consumption of power does not simply happen, but rather, information needs to be relayed between subsystems to simulate the exchange of power for wheel speed. In order to maintain a separation between the two subsystems so that the system can remain truly modular, an ontology must be developed for communication between the subsystems as mentioned in section 1.2.1. This is accomplished in the Horizon Simulation Framework through "dependencies" which allow data to flow between subsystems. In HSF, a dependent subsystem is a subsystem that must provide information to the subsystem that has a dependency on it. In the current working example, the ADC subsystem is a dependent subsystem of the power subsystem. By contracting the dependencies in this way, the omniscient user can provide dependency functions that allow subsystems to relay information that is understandable to the requesting subsystem. In HSF, this omniscience is carried out by maintaining the dependency functions in a globally accessible list and informing the subsystem which functions to call in their dependency collector.

Subsystem Node In HSF v2.3, in order to incorporate the subsystems with the dependencies in a multi-asset use case, a wrapper class was made to encapsulate the subsystem. The justification behind this wrapper class, according to O'Connor, Mehiel, and Butler, is that it was modeled after an adjacency graph in which the nodes are containers for the functional elements inside them in order to provide "plug and play" capabilities. This wrapper class is called a "subsystem node" and it holds a subsystem as well as the "node dependencies" which is a collection of dependency functions that the subsystem needs to execute its CanPerform function to update its state. This extra level of abstraction requires that the software has the functionality to correlate the subsystem to its subsystem node and dependency functions. Another important feature of using the subsystem node within the adjacency graph is that it can allow for recursion in evaluating subsystems. "Using this network structure and recursion in calling

subsystems to perform, subsystems automatically confirm that their predecessors had previously executed, and that any data that they might need had already been set to the [current] state" (O'Connor, Mehiel, and Butler, 2008). However, after an audit of the C++ code, it was discovered that recursion was being used in a loose definition of the term.

Node Dependencies Because a dependency function can return any type of profile (integer, bool etc.), there needed to be a way to store a reference to all different types of functions by return type for both hard coded and scripted dependency functions. The node dependencies class does this by maintaining a map of function call keys (strings) to function pointers, one map per return type. However, this use of multiple maps is wasteful as quite often there will be nothing stored in most maps (such as the bool or integer return type maps) and many things in one (like the double return type map). With the node dependencies tracking this information, a subsystem node need only to know what call key to use in order to get the correct dependency function, and the node dependencies class would do all the work to search through all the maps to find the function with the correct call key. While this solution was elegant from the perspective of the subsystem node, the implementation within the node dependencies class was bulky and repetitive. For instance, in order to call each different type of dependency, there was a 60 line function copied and pasted for each return type with the types simply changed.

Dependencies In keeping with the idea that dependency functions are an interface between subsystems, HSF v2.3 has a class specifically for maintaining all the dependency functions and dependency collectors. A pointer to these functions is what is maintained in the node dependencies class. The dependency functions require state information. From the state, the dependent subsystem extracts its own state information and transforms it before relaying it to the requesting subsystem. The dependency collectors call all the dependency functions for a specific subsystem and combines the returned information as appropriate. As a concrete example, the power subsystem might have a dependency on the EO sensor and COMM subsystem. The dependency collector would call the dependency functions for each of the dependent subsystems

then combine the information by adding the radio's and EO sensor's return value before returning it to the CanPerform method to be subtracted from the state. Since the dependency functions in HSF v2.3 accept a state, the dependency collector is also responsible for passing in the correct state. And since states are stored by asset, the dependency collector must also know which asset to get the state from. This is problematic when remembering that the CanPerform method is only passed the state for the asset which the subsystem belongs to. This means that the state for the dependency functions that are cross-asset are not reachable from the parameter and must come from someplace else. In order to ensure the dependency has access to the states from all the assets and not just the one being passed in through the CanPerform, the dependency class has a field called endStates, which is a list of the most recent states by asset (where the zeroth element of the list correspond to the first asset, etc.). The endStates field is updated once per simulation time step (or event) in a method called updateStates.

Update States After all subsystems have updated their state via their CanPerform and the simulation time step has completed, the updateStates method is invoked on the dependency class in order to pull in the most up to date states for the dependency collector. However, because this function is only called once per simulation time step, as the subsystems are updating their states within the event, the endStates field will still hold the old information from the end of the previous time step. Returning to the previous dependency example, the power subsystem must know how much power the EO sensor and COMM subsystems consumed in order to update its own state. However, the dependency function will be referencing the endStates field from the dependency class so if the EO Sensor has updated its state since the event started, then the power subsystem will be referencing old data from the last time step. This is considered an error and will be updated in v3.0.

2.3.4 Asset Schedule

In HSF v2.3, the schedules were generated on a per-asset basis. Figure 2.9 illustrates how asset schedules were constructed within the system schedule in v2.3. The events and states of all the subsystems in the asset were held at an asset level rather than a

system level. While this was an easy and obvious step from the previously single asset version of the framework, it didn't allow for state information to be relayed between assets at the instance that the state was updated by the subsystems' CanPerform methods because the assets all report to a separate state. The previously discussed updateStates method was created to allow subsystems to have dependencies to assets external to their own. The downside of posting state information to a system level only once per iteration is that subsystems will always be referencing data from the old event, even though their dependent subsystems may have updated their states within the current event. The fundamental problem was determined to be derived from the fact that the state is stored on an asset level, and could be solved by holding the state at the system level and having all subsystems report to the same state, regardless of what asset they're in. The implementation of this solution is discussed in section 3.4.

FIGURE 2.9: Asset Schedule in HSF v2.3 (O'Connor, Mehiel, and Butler, 2008)

2.3.5 Scripting

Language Popularity v2.3 used the Lua scripting language to incorporate customizable features into the functions of the modeling segment. While the Lua language is fast and lightweight (Ierusalimschy, Figueiredo, and Celes, 2016), it is not as common as Python, the most common scripting language according to Cass. In the IEEE article, Cass explains how Nick Diakopoulos "weighted and combined 12 metrics from 10 sources (including IEEE Xplore, Google, and GitHub) to rank the most popular programming languages" (Cass, 2014) and produce the infographic in Figure 2.10. It can

be seen that Lua ranks far below Python, the most popular scripting language.

FIGURE 2.10: Most Common Programming Languages Cass, 2014

While language popularity may not seem important from a programming perspective, it is important from a learning perspective. In the coming years it is anticipated that the people who will be working on HSF will be primarily aerospace engineering students with little to no programming experience. It is not untenable to argue that these students will be self-taught, and that more resources will be available in the form of documentation, examples and forums for a more common language than a less common one. In fact, this point is reiterated by Perkel. In his article, Perkel analyzes how Python is used in scientific computing by students and professors who are not inherently programmers by discipline. Titus Brown, a professor in Bioinformatics at Michigan State University, and Aerospace Engineering students with limited programming exposure are examples of the population Perkel speaks to. "The difficult part of learning to program lies with the fundamentals, says Brown—once a researcher has those nailed down, adapting to a new language is just a matter of syntax. What matters most in the early stages is having a good support network. 'Pick the programming

language based on what people around you are using,' Brown advises. Increasingly, that language is Python" (Perkel, 2015).

Incorporation into HSF In the C++/Lua version of the program, the Lua scripts are bound to C++ functions on a function by function basis. This method of incorporating scripting is limiting to the customization of the model as only certain exposed functions and features can be scripted. The user must also provide a reference to each scripted function in the XML file that initializes the model, making the process tedious and time consuming. A new, simpler approach will be taken towards scripting in HSF v3.0 that allows the user to inherit a subsystem and modify the object as needed.

Chapter 3

IMPROVING THE HORIZON SIMULATION FRAMEWORK

3.1 Motivation

The choice of a programming language for a project is typically dependent on the objectives of the software. However, in the case of Horizon, the ability of the developer was considered with more weight than the software goals. Because Horizon is a framework for improving the aerospace design process, it seems fitting that aerospace students should work on the code base in order to not only gain a better understanding of systems engineering design, but to also learn how to program in an object oriented language. While section 1.2.5 discusses results from one study that quantitatively found that C# is better as a first programming language than C++, even most versed programmers would agree that C++ is one of the more complicated programming languages to learn, more so than C#. As a push to take advantage of new software development technologies and to make the framework more accessible to aerospace students, the Horizon Simulation Framework will make a shift to C# from C++, and the more common scripting language of Python will be favored over Lua.

As a result of changing languages and a thorough comb-through of the code, improvements were made to HSF v2.3 in order to adapt to and take advantage of C#, the .NET framework, and the incorporation of Python. The sections that follow detail the perceived benefits as a result of this switch, and the modification and improvements that were made to the existing framework.

3.1.1 Integrated Development Environment

As mentioned in preceding chapters, one of the main goals of converting the framework to C# is to allow students without a major programming background to work on the project. C# is very tightly bound to its Integrated Development Environment (IDE), Visual Studio, which is a Microsoft product for developing software. Visual Studio is a key component in helping to accomplish the mission of creating a code base that is easy to adapt to.

IntelliSense Visual Studio provides the developer with helpful hints as far as why some code might be wrong and also helps prevent the developer from making errors via IntelliSense, its auto completion feature. IntelliSense allows the developer to have a rough idea of what he wants to do, then displays all possible continuations of the command. Figure 3.1 shows what is displayed when the developer is trying to instantiate the "SubsystemDependencyFunctions" dictionary within the ADCS subsystem. IntelliSense exposes the type of the property to the developer, as well as how to instantiate it with the tab auto-complete, and even that there is an error on the next line because the previous line is incomplete.

FIGURE 3.1: Microsoft Visual Studio IntelliSense

This is especially useful when the developer is adding additional functionality to the framework using existing functionality. The developer need only know what project to work in and where to find additional classes he may need, then he has access to all the documentation, methods and properties simply as he starts typing. However, much of the elegance and ease of programming in C# is lost when not utilizing Visual Studio as the development environment, and thus, C# is typically not used in scenarios where Visual Studio is not being utilized. This is not necessarily the case with other programming languages such as C++ that can be maintained, compiled and run with a few command line tools. At the time that HSF v2.3 was being developed, Visual Studio was not a free product, but that has now changed. With the release of Visual Studio

2015 Community Edition, C# became more accessible to many programmers because the new IDE is free to all. Not requiring students to buy software is very helpful for increasing involvement and accessibility to the software. It should be noted that Visual Studio is currently a Windows only application, so only Windows users can take advantage of it. While an operating system specific application may decrease involvement among non-Windows users, the release of the cross-platform text editor, Visual Code, may mitigate for this. Overall, the advantages that the Visual Studio IDE provides to developers were considered more profound than the disadvantages of being a Windows only project.

Integration with GitHub In order to facilitate contributions from a group of developers, the Horizon code base is hosted on GitHub.com, a source control service provider. Because the Visual Studio IDE seamlessly supports integration with GitHub, developers need not even learn git powershell commands, but can simply use the source control panel provided in the IDE. Within an hour long meeting, a group of 10 aerospace engineering students who had, for the most part, not seen object oriented programming before were able to install Visual Studio, download the environment from github, and start writing code. Keeping in mind that most new developers will not write perfect code the first time, each developer creates their own branch of the project that they can then merge back into the master branch once the code is proved to be working. Later sections will discuss how unit testing and GitHub integrated quality control services will be used to ensure checked in code complies with an agreed upon standard.

Building the Solution Among many other qualities of Visual Studio that make it easy to use, one especially important feature is the automatic maintenance of the .csproj file and the simplicity of building the program. The .csproj file is automatically generated by Visual Studio to maintain the reference paths for the project dependencies as well as other important build information. Rather than compiling and linking source files with a custom made makefile, Visual Studio allows the user to simply click the start button because it automatically maintains information that would normally be stored in a makefile. This seemingly trivial convenience is important to HSF because it emulates the environment most aerospace students are accustomed to programming in, MATLAB™.

3.2 Reverse Engineering from C++

Previous versions of HSF lacked an essential element of software design– the specifications document. Because of this, the new HSF v3.0 was created by reverse engineering functionality requirements from the C++ codebase. The requirements for HSF v3.0 were created by combining the functional requirements from C++ as well as the changes outlined in the remainder of this section.

3.3 Utilizing C# and .NET

Another advantage of C# is the ease of access to the .NET framework. For aerospace engineering students who are accustomed to the built-in functionality of MATLAB™, the libraries of the .NET framework are similar in that the Microsoft Development Network (MSDN) documentation is consistent, contains many examples and is centrally located on the MSDN website. As previously mentioned, the .NET framework provides functionality that the previous version of HSF had to either customly create or rely on the C++ standard template library for. The replacing of the old libraries with existing objects and methods from the .NET framework is outlined by this section. In almost all cases, the amount of user lines of code was significantly reduced by using the .NET framework, and many previously handwritten classes could be left out completely.

3.3.1 Iterators versus IEnumerableables

The C++ implementation of the framework utilized the Standard Template Library (STL) for generic classes, such as the vector, and iterators. The vector class combines the functionality of a random access array as well as a last in first out (LIFO) queue. The vector also has an iterator in order to make iteration simpler. In C#, the vector is replaced by a specific type of Collection depending on the desired access capabilities. A Collection that implements the IEnumerable interface in C# makes iterating even easier with the use of the foreach statement. Classes that implement the IEnumerable interface include the list, stack, queue and even dictionary. Rather than constructing an iterator before every for loop then incrementing an iterator pointer, the developer simply has to use foreach which is available with every IEnumerable class. Because

the STL vector class encompasses functionality that is dispersed among the C# list and stack, a dichotomy was developed to determine when to convert a vector to a list and when to convert to a stack. Essential, is a vector utilized the push or pop methods anywhere throughout the C++ code, it was converted to a stack. Otherwise, it was converted to a list.

3.3.2 Pass By Reference

A major difference between C++ and C# is in the way that values are stored and passed between objects and methods. While C++ has the ability to obtain the reference to any variable via the pointer, C# has two types, value types and reference types. Value types are similar to what would be primitives in C++ and all other objects are reference types. This means that all methods in the new code base that used custom objects would be passed by reference. Luckily v2.3 predominately utilized pass by reference rather than pass by value so the conversion was, for the most part, simple. However, at one point within the scheduling algorithm it is required to make a safe copy of the schedule so that the new schedule can reference the old schedule. In C++ this could be easily accomplished by simply giving all the new schedules that required the old schedule a copy of the same pointer. In C#, however, manipulating raw pointers is not common practice as pointers are not directly exposed to the developer. Because a schedule is a stack of events and both stacks and events are reference types, if an event was added to one schedule, it would be added to all other schedules that were created from the same schedule because they all are made from a shallow copy. This obviously is problematic as it is not what the scheduling algorithm says to do. The scheduling algorithm (Algorithm 1) requires that a subset of events, or subproblem, is only stored in memory once, and that all extensions to that schedule reference the same previous events. After reading the MSDN documentation on stack constructors, a solution is found to this problem. The copy constructor for the stack takes in a stack and creates a new stack that has a unique reference, but maintains a reference to all the objects in the original stack. This allows for the copied stack to be manipulated (i.e. Push and Pop) while not modifying the original stack. However, because the elements within the stack are all still shallow copies to the original events, if an event is modified within one stack, it will be modified in all the stacks that copied it. (Recall that an event that

has already been added to a schedule will never be modified because it has already been evaluated to its final state.)

3.3.3 Object

All C# objects inherit from the Object base class. As a result, anything can be stored as an Object. This is especially advantageous when abstraction is necessary or a function's return type is not known at compile time. C++ does not utilize a common base class so abstraction is not possible unless the user explicitly implements it. Dependency functions are a specific example of when the object class would have been useful in HSF v2.3. Because the return type of the dependency function can be any type of HSFProfile, the SubsystemNode class had a map for all types of dependency functions based on their return type, as elaborated in Section 2.3.3. However, with the use of the object as a return type and the implicit cast, the entire class was able to be eliminated with the caveat that the omniscient user casts the return value of a dependency function to its proper type.

3.3.4 Searching with LINQ

Microsoft provides an extension to the capabilities of the C# language with Language-Integrated Query (LINQ) (MSDN). LINQ allows SQL like queries on IEnumerable objects that reduces the need for searching with iteration. The code can be made shorter and more readable by replacing for loops that are searching through a collection with a single SQL like command.

3.3.5 Nuget Packages

For functionality that is not built into the .NET framework, Visual Studio provides an easy way to manage Nuget Packages to extend capability. Nuget Packages are user-managed libraries that Microsoft hosts but often does not maintain. One example of a Nuget package that is widely accepted, but not managed by Microsoft, is the Newtonsoft.Json package which provides functionality for Json operations. A Nuget package utilized by HSF is the IronPython package which is managed by Microsoft and the IronPython Community.

3.3.6 Built-in XML

The parsing of the XML input files was made drastically simpler with .NET XML library. Rather than creating a custom class to store and manipulate XML data objects, the .NET library for XML simply had to be imported and all the functionality was available to parse the files.

3.4 Architecture Changes

Keeping in mind that part of the reason for switching to C# is to make it easier for future developers to learn and improve the code base, it logically follows that the architecture should also be easy to understand. With the code base already being transformed into a new language, it is a good time to restructure the architecture to reflect the evolution of the framework as well as to create a baseline architecture that would encourage future evolution without a major restructuring. One of the major issues with the v2.3 architecture is that the code was highly dispersed and not necessarily organized due to the previously mentioned issue of simply tacking on new code. The new module decomposition and dependency structure of HSF is pictured in Figure 3.2.

FIGURE 3.2: The New HSF Architecture

Some of the modules were fundamentally changed in order to improve the functionality and learnability of the codebase. The following sections discuss these changes in detail.

3.4.1 Subsystem, Dependency, Subsystem Node, Node Dependencies

One of the most significant modifications to the previous implementation of HSF was in the way that subsystems were defined and interacted with each other. Key features of the .NET framework allowed for an architectural change that removed unnecessary abstractions within the code base. The subsystem node, whose only function was to communicate node dependencies to the subsystem was eliminated by moving the functionality to the subsystem itself. The node dependency was also eliminated as a result. This was all made possible with a major change to the dependency class. Previously, the dependencies were all stored in a C++ container called a map. A map is essentially an array of key value pairs that permits access to element within the array via a unique key. In Horizon, the map correlated function call keys (strings) to a function pointer. In order to accommodate for the variability of return types of dependency functions, there was one map of call keys for each return type as shown in Figure 3.3.

FIGURE 3.3: HSF v2.3 Dependency Maps

The issue of the variable dependency function return types is solved in C# with the object type as seen in Figure 3.4.

FIGURE 3.4: HSF v3.0 Dependency Dictionary

The documentation for the C# object explains that the object is "the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object"(MSDN, 2016). This means that variable return types of the dependency function is no longer a limiting factor because all dependency function returns can simply be cast to their correct type after the function is called. The result of this is a single dictionary of call keys to delegate functions– C# 's function pointer– in the dependency class. Requiring that all dependency functions be defined in the dependency class limits the developer from defining dependency functions elsewhere, like in a Python script. For this reason, the global omniscience of the dependencies is achieved by having a singleton instance of the dependency class at the program level. This one instance will hold the dictionary of all the dependency functions and their call keys and a dependency function can be defined anywhere so long as it is added to the dependency class.

The old dependency class also held the function definitions for all the dependency functions and collectors. While having all the dependency functions separate from the subsystem definition is consistent with the idea of the omniscient designer, it is unnecessary in the C# version of the program and with the new architecture. In HSF v3.0, the dependency functions are defined in the subsystem class and each subsystem constructor takes in the dependency instance and adds the dependency functions that it has to it. This allows dependency functions to be overwritten or newly defined in scripted subsystems. The standard followed by HSF v3.0 is to have the dependent subsystem provide the dependency function to the subsystem with a dependency on it. This approach was chosen because subsystems can only access their own state data with their StateVarKeys. By requiring subsystems to post their dependency functions to the global dependency dictionary, the dependency class is populated as each subsystem is instantiated. Then, the subsystems can find the dependency functions they require because the XML model input file informs the subsystems of the call keys that belong to them in the dependency class. So rather than have a subsystem node and node dependency class, the subsystem now just has a list on dependent subsystems and dependency functions in the form of delegate functions. This new method is so robust that it is also now used to communicate state data to things other than a subsystem such as a schedule evaluator or constraint evaluator. This maintains the standard

throughout the entire framework that only subsystems are allowed to access their own state data and the data can only be exposed via a call to a dependency function which acts as a contract between two entities.

The New Abstract Subsystem Class Because many of the subsystems require much of the same functionality, an abstract subsystem class was determined to be more useful than a subsystem interface. The abstract class provided two desirable traits: the subsystem class cannot be instantiated and generic functionality can be defined and then used or overwritten by the inheriting class. Table 3.1 provides a detailed description of the functionality that is built into the subsystem class.

TABLE 3.1: Subsystem Default Functionality

CanPerform	The default CanPerform method simply loops through all the subsystems in the dependent subsystems list and calls their CanPerform method. This ensures that all subsystems are evaluated in the order specified by the XML input file. It also guarantees that when the dependency functions are called, the dependent subsystems have all updated their states to be current with the new timestep. Every derived subsystem must call the base CanPerform method before continuing with its own CanPerform functionality. The abstract CanPerform then also sets the private state and task fields of the Subsystem so the deriving Subsystem doesn't need to search through the dictionaries of the Event.
CanExtend	The default CanExtend method simply checks the time to which the state needs to be extended to, and if the time is beyond the current event, the new event end is set to the extended time.
CollectDependencyFuncs	CollectDependencyFuncs is called in the main program after the dependencies have all been parsed from the XML document. The dependency collector takes in a list of call keys and uses each call key to index into the dependency class and copy all the dependency functions to a local list of dependency functions. With the dependency functions stored locally in the class, the dependency class need not be passed around throughout the entire program.
DependencyCollector	The dependency collector invokes and sums up the results of all the dependency functions that are stored within the subsystem.
GetSubStateAtTime	Because subsystems can only access their own state, the GetSubStateAtTime method returns the subsystem's state at a given time. This is primarily used when writing the schedule out to a file because state information should always be relayed from subsystem to subsystem via a dependency function.

3.4.2 State History, System State and Event

The Asset Schedule was replaced by the State History in the new version of HSF because it was unfavorable to have each asset have its own schedule as this meant they couldn't easily share state information. However, the asset schedule maintained important information about the time history of events. The State History is identical to

the asset schedule in that it also stores this information, however, it is not correlated to a specific asset, but rather the entire system. Similarly, rather than maintaining one System State per asset, the System State was held at a system level so that the subsystems could access any other subsystem, regardless of the asset it is associated with. An illustration of the State History class is provided in Figure 3.5.

FIGURE 3.5: State History Class that Replaces the Asset Schedule

This change was necessary in order to allow cross-asset dependencies without having a delayed updating of the states as mentioned in Chapter 2. However, switching the System State to be at a system level posed a problem for scheduling. Besides from the state of the subsystems, the System State also holds the start and end times for the task and event that System State is associated with. But if there is only one System State, then all assets must be on the same event and task schedule. This discourages multitasking within the system and does not accomplish the goal that HSF seeks to achieve. In order to allow each asset to be on its own event and task, the event and task start and end times were moved from the System State to the event class. The Event now contains the dictionary that correlates the asset to its respective task as well as additional dictionaries to maintain task and event start and end times.

3.5 Class Factories

The subsystem class that has many, more specific, classes that derive from it is and instantiating all the specific classes becomes tedious as their specific constructors must be called even though they are all of the same base type. In the 2.3 framework this was dealt with in the main method with clunky, hard coded conditionals, and an even less robust adapter class for managing more complex derived classes. Due to the high

variability in different types of subsystems, constraints and schedule evaluators that can be either scripted or created from the classes included in HSF, a class factory design pattern was utilized to instantiate these classes at runtime. According to MSDN, a class factory is simply a class that statically interprets an input and returns a specific instance that is of a derived type of the base class. By utilizing class factories, the XML interpreter only needs logic to interpret the XML, then the burden of actually instantiating the class is placed on the class factory which understands how and when to communicate with each of the derived classes' constructors. Figure 3.6 shows how a class factory can be used to consume a resource that is not useful to the system, such as an XML node, and create something that the system has an understanding of.

FIGURE 3.6: Class Factory Function MSDN, 2016

3.6 Incorporating Python

3.6.1 Choosing IronPython

Section 2.3.5 explains the motivation for switching to a more commonly used scripting language, namely, Python. However, there are many different "flavors" of Python. IronPython is a flavor of Python that runs on the .NET framework. While it lacks some of the classic libraries of the more common version of Python (Python that runs on C and is compiled to binaries), the ability to draw on the .NET framework provides all the same functionality and more (Wucher, 2010). IronPython is tightly bound to C# and provides the customize-ability that the user needs to specify his system. Wucher of MIT advises that use of IronPython with C# is best suited in cases when "delivering a business application to your end users... [and] they each have a custom processing need (e.g. business logic). Your application is the engine that contains and performs the main operations on the data." The goal of Horizon to provide customizable, robust, functionality to the user aligns very closely to this description. The remainder of this section will discuss the significant considerations for incorporating Python into C#. Python and IronPython will be used interchangeably as IronPython is a subset of Python.

3.6.2 Communication Between C# and IronPython

Because IronPython is dynamically typed, it needs a dynamic environment to attach to. The Common Language Runtime (CLR) is a dynamic runtime environment provided by the .NET framework that allows for many features including cross language integration and exception handling (MSDN, 2016). Through the CLR, the Python code can modify C# objects. Just as C# classes need references to classes that are not in their own namespace, the Python classes also need to know about classes that may be used in the Python code. This is preformed by adding a CLR reference to the Dynamic-Link Library (DLL) that contains the class specification. Figure 3.7 shows how the references to .NET and Horizon frame work are added.

```

1  import sys
2  import clr
3  import System.Collections.Generic
4  import System
5  clr.AddReference('System.Core')
6  clr.AddReference('IronPython')
7  clr.AddReference('System.Xml')
8  clr.AddReferenceByName('Utilities')
9  clr.AddReferenceByName('HSFUniverse')
10 clr.AddReferenceByName('UserModel')
11 clr.AddReferenceByName('MissionElements')
12 clr.AddReferenceByName('HSFSystem')

```

FIGURE 3.7: IronPython Code to Add References to .NET and HSF libraries

3.6.3 Inheriting a C# Object

In the previous version of HSF, the scripting aspect was performed on a function by function basis as mentioned in 2.3.5. This method of incorporating scripting results in increased overhead of binding scripted methods to function calls within the framework and limits the user to only customizing exposed functions. IronPython and C# provide an alternative to this that is not possible in Lua. In Lua, there is no concept of classes like there is in Python. Because both IronPython and C# are built on .NET, it is not difficult for a IronPython object to inherit from a C# class that is exposed in a DLL. This has many benefits for the HSF user. If the user likes the default implementation of some of the methods of an already implemented subsystem, but wishes to change

other methods, the Python class simply needs to inherit from the built-in subsystem and override the implementation of the methods. Similarly, if the user wants to create a brand new subsystem that is not currently in the framework, he can inherit from the abstract subsystem class and define the new functionality from the ground up. Details about how to create custom Python scripts are included in the user manual found on the GitHub repository.

3.6.4 Strongly Typed versus Loosely Typed

One of the biggest differences between C# and Python is that Python is a loosely typed language while C# is a strongly typed language. This is primarily an issue when passing arguments to a Python method from C#. However, because the Python class has access to the C# DLLs, it can implicitly convert the arguments to their appropriate type at runtime. Unfortunately, if this conversion fails, a runtime error will occur.

3.6.5 Pass By Reference versus Pass By Value

Another big difference between Python and C# is that Python is pass by value while C# is pass by reference. This would seemingly have a large impact on being able to use Python as most C# methods rely on pass by reference to update an object because C# only allows one return value. However, the *IronPython Documentation* explains that IronPython implicitly converts to a reference type when necessary to allow for the modification of object parameters.

3.6.6 Runtime Errors

As mentioned in the previous sections, the use of Python can cause runtime errors. Runtime errors are especially hard for the user to debug because the information provided typically isn't relevant to a user that is unfamiliar with what is happening behind the scenes in the C# codebase. In order to limit the occurrence of runtime errors, the Python class is encased in the scripted subsystem class. This C# class performs pre-processing of the data before it is passed to the Python class in order help mitigate runtime errors, and throw more useful exceptions to the user.

3.7 Uniform Coding Style

As the code has been rewritten, an adherence to a uniform coding style was relatively easy to maintain as there were only two developers who both had an understanding of style rules. The uniformity that currently exists will be beneficial to maintaining the codebase as it creates easier to read code that can be adapted to quickly as discussed in section 2.3.1. In order to maintain this standard of quality, the new developers will be provided with style guidelines that can also be found on the GitHub repository.

3.8 Unit Testing

The utilization of testing was not apparent in the codebase of HSF v2.3. If unit testing occurred, it was perhaps eliminated for the purpose of releasing the code. In order to stay true to the guidelines of test driven software development, unit testing, or some version of incremental testing, should be employed throughout the development process. Visual Studio provides a simple way to add unit test projects to a solution. These test projects persist in the software but are automatically not compiled into the release configuration of the solution. The unit tests automatically populate the test explorer (Fig. 3.8) and can be run individually or all at once for a comprehensive check of the software. Every project within HSF has its own unit testing project.

FIGURE 3.8: Visual Studio Test Explorer

Chapter 4

SIMULATION RESULTS

4.1 The Test Cases

There are 4 test cases for HSF, two of which are used to test throughout the development phase and two of which are used to validate the final product once the development phase is considered to be done. The results from the final development test case will be discussed in the remaining sections of this chapter. The double asset model is known as the Aeolus subsystem that is discussed in 2.3 and the dependency tree is found in 2.1. The subsystems that makeup the Aeolus test case can be found in 2.1

The simulation can be tested as a static system or a scripted one. The static test case runs completely on the built in HSF model and the scripted one relies on the developer to provide Python scripts to simulate the scripted behavior for subsystems, dependencies, equations of motion and schedule evaluators. These test cases are the same as the test cases used in HSF v2.3 in order to validate that both versions produce similar results as v2.3 is the guidance of correctness for v3.0.

4.2 C++ Baseline Results

The C++ framework found a schedule that captured 271 targets total, with a dispersion showed in Figure 4.1. State data simulation results from HSF v2.3 are documented by O'Connor, Mehiel, and Butler. Figures 4.2 and 4.3 show the state data found for data collection and power functions of the system. This data was achieved by running HSF

FIGURE 4.1: HSF v2.3 Targets Hit

with the Aeolus model and no scripting, keeping 2 schedules at every time step and outputting the best schedule at the end.

FIGURE 4.2: HSF v2.3 Payload Data Subsystem State Data

FIGURE 4.3: HSF v2.3 Power Subsystem State Data

4.3 C# Results

The C# program also found a successful schedule for both the assets to image their ground targets within the constraints and specifications of the system. HSF v3.0 found a schedule that captured 307 images and is pictured in Figure 4.4

FIGURE 4.4: Targets Hit in HSF v3.0 Optimal Schedule

Figures 4.5 and 4.6 show the relevant data that comes into the system via the pixels recorded by the EO sensor, the data held by the system in the SSDR, and the data downlinked by the system via the radio. Figures 4.7 and 4.8 illustrate the details of the system power. The power state of the EO sensor, battery depth of discharge and solar panel power in and all shown in a side by side comparison. In both assets, it can be seen that when the spacecraft is in sunlight, the solar panels offset the power consumption by the EO sensor and radio.

FIGURE 4.5: Data Buffer Fill Ratio Considering Number of Pixels in and Data Rate Out for Asset 1

FIGURE 4.6: Data Buffer Fill Ratio Considering Number of Pixels in and Data Rate Out for Asset 2

FIGURE 4.7: Battery Depth Of Discharge Considering Solar Power In and EO Sensor Power State for Asset 1

FIGURE 4.8: Battery Depth Of Discharge Considering Solar Power In and EO Sensor Power State for Asset 2

Section 4.1 discusses the various test cases used to validate the results of the new HSF implementation. These same test cases were used to profile the performance of the C_‡ framework. The static test case in the C++ simulation was found to take about 5 hours by O'Connor, Mehiel, and Butler. Visual Studio Community 2015 provides a useful tool for profiling program performance called the Performance Wizard. The Performance Wizard is capable of exposing the most used and most time consuming methods as well as overall runtime and CPU usage. Figure 4.9 was generated from data provided by this tool within Visual Studio. The profiling was run on a Intel® Pentium® 2.10 GHz processor with 3.71 GB usable RAM.

FIGURE 4.9: HSF v3.0 Run Times by Test Case

Not surprisingly, the static test case is the fastest, but by only about 30 seconds over the scripted EOMs version and 2 minutes over the scripted evaluator.

Chapter 5

FUTURE WORK AND CONCLUSION

5.1 Expanded Schedule Diagnostic Capabilities

While the HSF framework accomplishes the goal of providing the user with information about their system, the framework itself does little with this information. Using methods of statistical processing, the framework could use state data to modify the order in which subsystems and constraints are evaluated in order ensure the system fails fast to improve runtime.

With state information of the failed subsystem at hand, the user might try to modify the subsystem to see how sensitive the entire system is to the failing subsystem. An automated sensitivity analysis could be incorporated into HSF such that when the failed schedules are analyzed and a subsystem is seen to be the most commonly failing subsystem, the framework interacts with the model to randomly cycle its parameters and measure the effect on the entire system.

5.2 Expanded Universe and Subsystem Models

Now that the baseline framework is established, more built-in functionality can be added in the form of more complex universe models and a wider range of subsystem models. Currently, the framework has a sun model that is used by the power subsystem and nothing else. Density, cloud cover, and even interplanetary models would allow the user more simulation capabilities. While this should be accomplished with built in models, adding the ability to script a universe model similar to how subsystems, equations of motion and schedule evaluators can be scripted would also be beneficial to the user. The subsystem models that HSF includes are minimal and lacking in variety. Currently there are rudimentary models for ADCS, Communication, Earth Observing Sensor, Power and the Solid State Data Recorder. While the subsystems can all be overwritten with Python, the more useful built in functionality provided by the framework, the easier it is to use for the user.

5.3 Multi-Threading

A brief attempt was made at creating a multi-threaded scheduler but time did not permit to accomplish this task. Future work could be done to add multi-threading to the scheduler and compare the performance results.

5.4 Graphical User Interface

HSF v2.3 had a prototype graphical user interface that is now obsolete because it is no longer interfacing with C++ and the XML formatting specification has changed. The prototype, however, is an elegant example of the vision of the Horizon Simulation Framework to be completely GUI based. Within Visual Studio, future work could be done to create a GUI that interfaces with the C# codebase, automatically generates well-formatted XML files and Python scripts from the user's design in the GUI and invokes the framework. When GUI development begins, an emphasis should be placed on carefully designing the interface between the GUI and the codebase. The interface will allow the two components to be managed as distinct entities that can change independently of one another. Once the interface is defined, however, changing it will be an extensive task.

5.5 Agile Development

As the team and codebase grow, it is the advice of the author to adopt agile software development. Such development allows for smaller chunks of the larger problem to be tackled in two week sprints with the result of every sprint an improved and working code base. The division of work into small chunks allows for the product manager to allocate work on a schedule that aligns well with the normal turn around time for academic assignments. The agile development paradigm can be researched at <http://agilemethodology.org> (*Agile Methodology*).

5.6 GitHub Quality Control

One feature of Github that has not yet been set up is the ability to interface with quality control services. These services provide a heads up display integrated into the Github repository that supply metrics like code complexity, documentation, build passing, test

case passing and more. Having these metrics automatically updated with every check-in gives the team immediate feedback of when something needs to be fixed and conforms to the continuous integration concepts addressed in agile software development.

5.7 Model Driven Development

The long term goal of the Horizon Simulation Framework should be to provide MDD functionality defined in Section 1.1.2. Once the HSF GUI is created, the user should be able to graphically define the model, run simulation and perform analysis, then automatically generate technical specification for the system elements and the interfaces they will interact on. The incorporation of MDD into HSF would make it a comprehensive tool that could be utilized through the Critical Design Review of an aerospace project.

5.8 Conclusion

At the time of the Horizon Simulation Framework's conception, Visual Studio and the .NET framework were novel utilities with far less capabilities than they have now. About 8 years ago, the Horizon Simulation Framework reached its most capable form, and around the same time, C# started to emerge as a viable method to implement object oriented programming. Unfortunately, around the same time, development on HSF came to a halt and the framework remained in the v2.3 iteration until Spring of 2016. By the time the program was revived, Python had emerged as a front runner to the choices of scripting languages and entire companies had been built on the .NET framework. In order to keep the Horizon Simulation Framework current with new software development technologies, an elegant solution was found that provided seamless integration of a C# backbone with Python scripting. The switch to C# also allowed for architectural changes to be made to the framework in order to create an easy to learn and maintainable codebase. The promising results presented by this thesis suggest a bright future for the progress of the framework. The baseline that has been created can be expanded in almost every aspect to continue to grow the framework's capabilities.

Although the focus of this thesis has been on the aerospace design process, and more specifically, the Aeolus mission, the Horizon Simulation Framework has the potential to be applied to every discipline of engineering that follows the fundamental

process of systems engineering. The prospect of the new models that can be imagined by the creative engineer and then simulated in HSF are exciting to the future of engineering design. Rough, unprecedented ideas can be quickly realized in the form of modeling and simulation to test the bound of what we previously thought was possible.

BIBLIOGRAPHY

- 15288-2008, Std (2008). *Systems and Software Engineering System Life Cycle Processes*. 2nd. IEEE ISO/IEC.
- agilemethodology.org. *Agile Methodology*. (Visited on 05/03/2016).
- Bass, L., P. Clements, and R. Kazman (2003). *Software Architecture in Practice*. 2nd. Reading, Mass.: Addison-Wesley.
- Bellman, R. (1954). "The Theory of Dynamic Programming". In: *Bulletin of the American Math Society* 60, pp. 503–515.
- Cass, S. (2014). *Top 10 Programming Languages*. (Visited on 04/26/2016).
- ESA (2016). *Virtual Spacecraft Design*. (Visited on 06/09/2016).
- Hawthorn, C.J., K.P. Weber, and R.E. Scholten (2014). "Reusable, Modular, and Scalable Flight Software". In: *2014 IEEE Aerospace Conference*, pp. 1–7.
- Ierusalimschy, R., L. H. Figueiredo, and W. Celes (2016). *Lua Documentation*. (Visited on 04/26/2016).
- IronPython Community (2016). *IronPython Documentation*. (Visited on 04/27/2016).
- Lew, A. and M. Holger (2007). *Dynamic Programming: A Computational Tool*. Springer.
- Loper, M. (2015). *Modeling and Simulation in the Systems Engineering Life Cycle: Core Concepts and Accompanying Lectures*. XIX. Springer.
- Maruping, L.M., XJ. Zhang, and V. Venkatesh (2009). "Role of Collective Ownership and Coding Standards in Coordinating Expertise in Software Project Teams". In: *European Journal of Information Systems* 18.4, pp. 355–371.
- MathWorks (2016). *SimEvents Documentation*. (Visited on 06/08/2016).
- MentorGraphics. *Questa Formal Verification*. (Visited on 06/08/2016).
- Micouin, P. (2014). *Model Based Systems Engineering: Fundamentals and Methods*. John Wiley and Sons, Inc.
- Moghaddam, K.S. and J.S. Usher (2011). "Preventive maintenance and replacement scheduling for repairable and maintainable systems using dynamic programming". In: *Computers and Industrial Engineering* 60, pp. 654–665.
- Morin, T.L. and R.E. Marsten (1975). "Branch-And-Bound Strategies for Dynamic Programming". In: *ORSA/TIMS national meeting* 60, pp. 17–19.
- MSDN (2016). *C# Programming Guide*. (Visited on 09/28/2016).

- O'Connor, C., E.A. Mehiel, and B. Butler (2008). "Horizon 2.1: A Space System Simulation Framework". In: *AIAA Conference Paper AIAA Modeling and Simulation Technologies Conference and Exhibit*.
- Perkel, J.M. (2015). "Pick up python: a powerful programming language with huge community support". In: *Nature* 518.7537, p. 125.
- Russell, S.J. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall.
- Selic, B. (2003). "The Pragmatics of Model-driven Development". In: *IEEE Software* 20.5, pp. 19–25.
- Shoaib, F.M. et al. (2014). "An Evaluation Framework and Comparative Analysis of the Widely Used First Programming Languages: E88941". In: *PLoS One* 9.2, pp. 1–25.
- Wertz, J.R. and W.J. Larson (1999). *Space Mission Analysis and Design*. Space technology library. Microcosm.
- Wucher, J. (2010). *Embedding IronPython in a C# Application*. (Visited on 02/09/2016).

APPENDIX: GLOSSARY

class The class is an object oriented construct for defining the properties and methods of an object to be used as a template. 21

dependency functions A functional interface between two subsystems. In HSF, the dependency function is provided by the dependent subsystem and performs unit conversion, if necessary, for the subsystem with the dependency on it. 27

dependent subsystem A dependent subsystem is a subsystem that must provide information to the subsystem that has a dependency on it. i.e. The solar panels are the dependent subsystem to the Power subsystem because the power subsystem has a dependency on the solar panels to provide information about incoming power. 27

developers A person who writes C# code for HSF. Typically a Cal Poly Student.. 25

HSFProfile The HSFProfile is a sorted key value pair storage system that hold state data based on a time based key. The HSFProfile is used within the SystemState to hold state data. 38

inherit (inheritance) In object oriented programming, inheritance is when one object is derived from a base, or parent, object. To inherit from a class means to adapt its implementation in order to extend it. 38

loosely typed The type of variables, parameter or return value are determined at run time. Loosely typed languages include Python and MATLAB™. 47

object oriented A modular programming methodology that allows objects to be made that contain both information about the object and methods to be used on the object. 4

singleton A design pattern for creating a class that only allows for a single instance of the class to be made in the program. 41

strongly typed The type of variables, parameter or return value must be defined at compile time. Languages that are strongly typed include C#, C and C++. 47

users A person who uses the HSF software product to create models and run simulations on their system. 11