TESTING AND VERIFICATION FOR THE OPEN SOURCE RELEASE OF THE HORIZON SIMULATION FRAMEWORK

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

William Jackson Balfour

November 2022

COMMITTEE MEMBERSHIP

TITLE: Testing and Verification for the Open Source Release of the Horizon Simulation Framework

AUTHOR: William Jackson Balfour

DATE SUBMITTED: November 2022

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

COMMITTEE MEMBER: Morgan Yost, M.S.
Software Engineer

COMMITTEE MEMBER: Pauline Faure, Ph.D.
Professor of Aerospace Engineering

COMMITTEE MEMBER: Dianne J. DeTurris, Ph.D.
Professor of Aerospace Engineering

ABSTRACT

Testing and Verification for the Open Source Release of the Horizon Simulation Framework

William Jackson Balfour

Modeling and simulation tools are exceptionally useful for designing aerospace systems because they allow engineers to test and iterate designs before committing the massive resources required for system realization. The Horizon Simulation Framework (HSF) is a time-driven modeling and simulation tool which attempts to optimize how a modeled system could perform a mission profile. After 15 years of development, the HSF team aims to achieve a wider user and developer base by releasing the software open source. To ensure a successful release, the software required extensive testing, and the main scheduling algorithm required protections against new code breaking old functionality. The goal of the work presented in this thesis is to satisfy these requirements and officially release the software open source. The software was tested with $> 80\%$ coverage and a continuous integration pipeline which runs build and unit/integration tests on every new commit was set up. Finally, supporting documentation and user resources were created and organized to promote community adoption of the software, making Horizon ready for an open source release.

ACKNOWLEDGMENTS

Thanks to:

- My parents for their encouragement, love, and financial and emotional support to push through this project.

- My love, Chella.

- A special thank you to Eric Mehiel, for mentoring me through this arduous process.

TABLE OF CONTENTS

LIST OF FIGURES

Nomenclature

$AADL$ Architecture Analysis Design Language

CD Continuous development

CI Continuous integration

$CONOPS$ Concept of operations

$COTS$ Commercial off the shelf

DAW Digital audio workstation

GUI Graphical user interface

HSF Horizon Simulation Framework

IDE Integrated development environment

$MAST$ Multidisciplinary-design Adaptation and Sensitivity Toolkit

$MBSE$ Model-based systems engineering

MDD Model driven development

$OSATE$ Open Source AADL Tool Environment

OSS Open source software

QA Quality assurance

RFP Request for proposal

SE Systems engineering

STK Systems Tool Kit

Chapter 1

INTRODUCTION OF MODEL-BASED SYSTEMS ENGINEERING AND AN OPEN SOURCE HORIZON

This chapter will discuss the general foundations of Horizon by giving a background on space mission design, systems engineering (SE), model-based systems engineering (MBSE), and model-driven development (MDD). The Horizon Simulation Framework and its use must be understood in this context when considering the requirements for pursuing an open source release of the software. The goal is to maintain the security and effectiveness of the tool while improving it with contributions from the greater developer communities.

1.1 Mission Design and Systems Engineering

Aerospace projects are unique in their scope and singular application. Most aerospace projects involve large teams of engineers, development cycles on the order of years, and massive financial backing. Therefore, analysis tools are commonly used in early stages of development to inform key driving decisions prior to pouring excessive time and capital into a concept. Some analysis tools may continue to be useful for many stages of development, from mission design to component design and finally system verification. This thesis discusses testing and usage of Horizon, one such design analysis tool.

1.1.1 Space Mission Design

The process of designing a space mission typically begins with identifying constraints and objectives. These define the qualities of the space system which are achievable within the time, budget, or other constraints. So that time and finance is spent efficiently, system characteristics are defined by only what the system needs to perform the objectives. At this stage of development, objectives, constraints, principal players, timescale and estimating high level quantitative parameters should be defined as fully as possible. These combine to form the requirements which will shall be the guide for every following design decision. Program designers must then define alternative mission designs as candidates for the final mission concept within these requirements. Sufficiently defined candidate mission architectures must then be thoroughly evaluated for performance and mission utility. This is where a program decides what the system is and what it does, which involves defining budgets like SWaP, ground/space support needs, etc. Then, the candidate is assessed for its ability to perform the broad objectives of the mission and fulfill the needs of the end user. At last, the mission designers must involve the customers, who are shown how well the alternative candidates satisfy the original objective vs cost.

Once a candidate is chosen, a baseline system design is constructed. From here, many individual parameters can be adjusted relative to the baseline simultaneously rather than assessing every combination of altitude, power, mass, etc. The baseline updates as parameters which perform better replace old baseline numbers. As it matures, the baseline becomes more rigid and eventually becomes the system design. System requirements should then be revised to reflect the original objective and the tools, systems and techniques available. At this stage, the alternatives should again be explored and iterated to see if any parts of discarded designs can be useful, if the objectives are rigid or if they can be altered to decrease cost or risk, or if an

alternative satisfies the objectives better. The mission design process finishes with the finalization of the Concept of Operations (CONOPS) which describes the processes that each asset in the system performs.

From testing and integration to end of life procedures, the CONOPS describes the how each system performs during all modes of operation. The CONOPS bridges the understanding between customer and the designers or similarly relates the request for proposal (RFP) to the formal requirements. Once the mission design is settled upon, the work of mission designer concludes, and the job as a system engineer begins. A system engineer must flow down requirements to the components level where the numerical requirements are satisfied by a product or group of products that will be delivered to the customer. Modeling and simulating the system prior to giving system requirements to the systems engineer provides a multitude of benefits which ultimately lead to lower program cost and quicker development.

1.1.2 Modeling and Simulation for Mission Design

Space mission engineering benefits greatly from rapid modeling and simulation. Low fidelity models are easy to construct and appropriate during early stages of development because the understanding of the details of the system is similarly low. Simulating low fidelity models provides a more detailed and accurate prediction of the behavior and performance of an unknown system than a back of the envelope calculation or an opinion of a single expert. With the objectives and constraints identified, engineers can model alternate candidate missions to quickly sift out ideas which don't satisfy objectives within the constraints. This technique can help a team converge on a general design or a few ideas, which they can immediately begin trade studies on, eliminating unnecessary work needed to investigate the non-feasible system or mission configuration. The team can then quickly move forward with a concept,

because they can be confident that a system exists which can perform the customers objective, without the need to build or test the system. A good model and simulation pair should be able to perform the following: quickly verify if a new requirement or objective can be met by the same system, describe the performance when fine tuning a parameter, or identify parameters which disproportionately hinder the performance of the system.

The Horizon Simulation Framework (HSF) is a modeling and simulation tool which is well suited for space mission design [1]. A mission designer can quickly model several alternative candidates and simulate them performing the mission. Horizon returns possible schedules for the candidate to perform the mission. If a candidate results in only one empty schedule, then the exhaustive algorithm could not find a solution for the candidate to satisfy the mission objectives. This candidate must be either removed from consideration or modified until a valid schedule is returned. Candidates which produce valid schedules can be directly compared using their schedule value, a sum of the frequency and weight of performed tasks. Horizon provides mission designers an idea of which mission concepts show promise and which proposals to discard or modify. For a more detailed look at Horizon's history, architecture design, and capabilities, see Appendix A.

1.1.3 Model-based Systems Engineering

Model-based systems engineering is the use of model-based engineering principles and best practices to applications of systems engineering [2]. INCOSE, a leading organization for systems engineering, defines MBSE as "an approach to engineering that uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle [3]." The purpose of systems

engineering is to approach extremely complex systems, often systems of systems, in a systematic manner to reduce the complexity. Therefore, MBSE must simplify complex systems through construction of models, and ultimately lead to successful systems which satisfy a range of customer needs [4].

In MBSE, the role of models is central to the engineering design process. The models become the authority in defining the specification, design, integration, validation and operation of the system [5]. After an RFP or other text-based specification sheet has been translated into objective, exactifiable metrics in the form of requirements, these measures are used to create models. These objective models replace the subjective text-based specification as the authority for requirements [5]. The requirements for a mission become embedded in the models of the system, providing consistency and efficiency through each stage of development.

Consistency is a feature throughout the MBSE process. Most broadly, MBSE uses consistent systemic approaches to complex problems, regardless of specifics or domain. There are three common iterative design approaches, the waterfall, the spiral, and the 'Vee' method. While everything regarding MBSE mentioned in this paper can be mapped to a stage on each design approach, the 'Vee' method, Fig. 1.1, is the assumed method for this report. MBSE promises a consistent language to describe the problem and solution, as well as consistent documentation produced at every step in between. When baselines are updated or a new phase of design is entered, the model remains largely the same, minimizing human error. Overall, MBSE reduces the complexity of complicated system of systems problems which reduces errors, increases efficiency and allows engineers to perform higher level analysis and design than previously possible.

Horizon is a modeling and simulation tool which supports systems engineering objectives. In initial design stages, it is used to quickly stand up alternative designs and

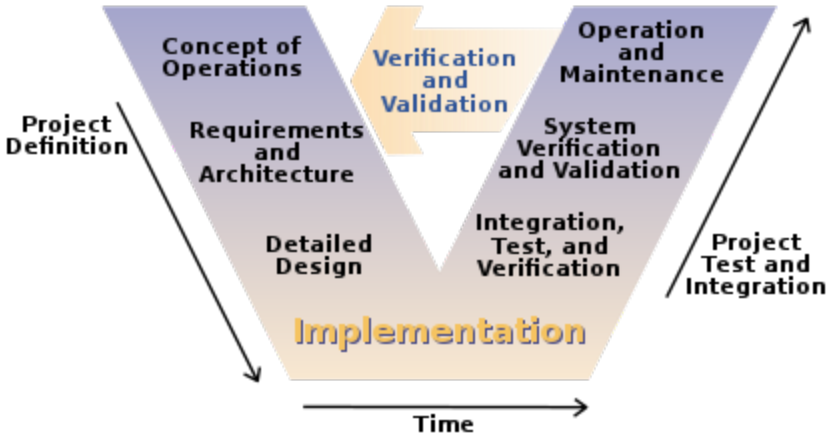

Figure 1.1: V or "Vee" model of the systems engineering process [6]

rank performance. After key characteristics of a mission are settled upon, Horizon can quickly simulate performance differences between similar missions with slightly different parameters or initial conditions. These models can be reused in later subsystem trade studies and design verification. The model may be continuously updated with baselines to confirm the system's ability to complete the mission. In Appendix A.1.5, Horizon's function as a model-based systems engineering tool is discussed in greater detail.

1.1.4 Model-driven Development

Model-driven development describes software development which focuses on developing abstract models of a system and then systematically transitions them into implementations of the system [7]. MDD is a key feature of the Horizon Simulation Framework. A defining characteristic of MDD is a primary focus on models rather than computer programs. Models are built with a greater focus on the real system in question, can be written and understood by domain experts rather than computer scientists, and are less sensitive to the computing technology and its evolution than

specifically written programs [5]. Domain experts are generally the best equipped to describe their system, so it follows that they should be most directly involved in modeling and simulating. Models are often much easier for experts to understand, interact with, configure and modify than software [5]. When domain experts are able to create the models themselves, the software developer, who can act as a barrier between ideas and realizations, can be mostly or entirely eliminated. This eliminates many obstacles like the possibility of mistranslation of high-level ideas, unintelligible software for the experts to adjust or otherwise use, and the software engineer being unavailable to answer questions or update software. Additionally, models, unlike traditional software, are much likelier to be written hardware independent. These features align well with HSF purpose, to assist in the development of long term, high complexity projects.

MDD becomes especially useful for long and complex design processes. Once a model is created to develop the CONOPS during Phase A, the model can be continuously updated and re-used in Phase B where baselines are optimized without redundant work remodeling the subsystems. If a team must make new models for the baseline, this adds extra unnecessary work and an unnecessary opportunity for human error. MDD adds tremendous efficiency when, after system fabrication, the models used to simulate and test the system are seamlessly transitioned onto flight hardware as the software [7]. Many modeling software products such as MATLAB's Simulink allow for the automatic creation of C programs from models, greatly reducing time, effort, and opportunities for mistakes. The output of the modeled system can be relayed to actual systems in the form of a set of sequential instructions or compiled, executable software for product hardware.

1.2 Open Source Communities and Advantages to Being Open Source

Open source software (OSS) describes software which can be freely used, modified, and shared. This gives the public permission to use source code or product design and promotes universal access to software. This opens the door for a symbiotic relationship between the user and developer community and the software. The software benefits from an open source community providing free feedback and development for the software. In exchange, users can have the software for free and modify it for custom purposes or build a new product on top of the open sourcesoftware. While the benefits to the software are a driving factor for choosing to go OS or not, the benefits to the user must be seriously considered during roll-out for the software to actually attract users and realize these benefits for the software. Below is the discussion about the benefits to the software and core development team. Benefits for the user is discussed at length in Chapter 3.

1.2.1 Overview of open sourceSoftware

The open source movement was started in response to the limitation of proprietary software, and spread quickly with among software enthusiasts. Communities of open source developers grew rapidly after the adoption of the open source concept by large projects like Linux and Netscape. The software is well suited to the audience it attracts and vice versa. The experimental and decentralized nature of open source-software cultivates a user base which is generally interested in tinkering, testing, and improving. And the software improves by attracting these users which are likely to introduce improvements and features or find and fix bugs. Because these users will always tinker, these software often innovate at a much higher rate than traditionally developed software. To this day, many innovations in the most widely used software

(e.g., browsers, word processors, IDEs, DAWs, image processing, etc.) originate from open sourceprojects prior to being adopted by the traditional industry leaders. Finally, users which participate in open source development are more likely to stick with the software that they've played a part in improving, which creates a loyal user base. When this feedback loop gains a critical mass of developer-users, cutting-edge, long-lasting, and stable products emerge like Linux, Git and MySQL.

1.2.2 Advantages to Being Open Source

For the software, there are many potential benefits of going open source. The largest potential benefit for a developer considering releasing their software open source is saving time. Especially for small teams or startups, time is the most valuable resource. Users become testers, providing quality assurance (QA), finding bugs, suggesting improvements, and determining fitness for the intended market. The free, ceaseless QA of an open source audience is the second major advantage. The founder of Linux, Linus Torvalds, famously said "Given enough eyeballs, all bugs are shallow." Going open source may lead to many genuinely interested eyeballs scrutinizing each line, which results in shallower bugs and better code. Conversely, code developed by a small, closed team might only have the original author and one disinterested QA, who's job necessarily delays deployment to catch bugs, contradicting the motive of for-profit companies, which is to release as soon as possible to maximize profits. The third benefit is transparency. When a code base is open source, anyone using or considering using the software may check the source code for errors and vulnerabilities, increasing the trustworthiness of software. Finally, a major reason to go open source is to share the work with the world. Many programs are made as passion projects and developers want to make a positive impact by offering free use of their software. These advantages must be considered against the drawbacks of open source software,

like vulnerabilities or bugs being introduced by malicious or ignorant developers, discussed in subsequent chapters, and loss of revenue by offering software for free.

Combining the MBSE with open source software development is of great interest for this paper. While there are many software packages which accomplish some sort of system modeling and analysis, they are also some combination of proprietary, closed development, confusing, and or limited in scope. Creating an MBSE tool which has its development driven by an interested user and developer community would be a useful and novel addition to the MBSE field. Horizon could fill this gap in MBSE, providing its time based simulation algorithm as a test-bed for developers and users to experiment with all types of models and applications.

1.3 The Future of Horizon as an Open Source Model-based Systems Engineering Tool

Horizon is a modeling and simulation tool which employs dynamic programming to assist in model-based systems engineering and mission design, analysis and verification. HSF has seen varying degrees of development for more than 15 years and its creators believe that its architecture is sufficiently mature to be released as an open source project. It is the hope of the creators that the open source communities will provide the benefits discussed in the preceding section. Specifically, the Horizon team would like to see a rapid expansion of the modeling library for new potential missions from the increase in general users, and an increase in development of new features and bug fixes from the increase in development inclined users. The goals of this thesis project focus on the final steps to prepare Horizon for an open source release.

1.4 Thesis Structure

Ahead, I will discuss the context of Horizon, providing background information on model-based systems engineering and model-based development and how Horizon compares to the current available tools in this field. Chapter two finishes with the motivation, barriers, and primary goals of this project. Chapter three discusses concerns of going open source and how continuous integration (CI) addresses these concerns as well as how to implement continuous integration effectively. Chapter four focuses the specifics of how I implemented Horizon's testing and continuous integration scheme. The fifth chapter discusses the results of this project, and what this project enables for the future of Horizon. The final chapter lists suggestions of projects for future students looking to develop for Horizon and what lessons I learned while performing my project for HSF. In Appendix A, Horizon's architecture is described along with a brief history of the Horizon project up to its current state.

Chapter 2

BACKGROUND ON MBSE AND OPEN SOURCE AEROSPACE SOFTWARE

An open source modeling and simulation and systems engineering tool must be created with a purpose. To understand the purpose of Horizon, it is necessary to gain an understanding of both Horizon and other projects with which Horizon co-exists. Surveying the landscape of open source projects in the aerospace field and their relative success is also important when deciding whether and how to join by going open source.

2.1 Barriers for adopting MBSE

To achieve the goals of adoption and community building, it is important to study how users view other MBSE tools. An online survey of systems engineers by Object Management Group (OMG), creators of SysML, probed the following questions: What are barriers to adopting an SE tool? What are the most and least used aspects of the tool? Does training increase the value of the tool? What is the hardest part of learning a systems engineering tool [8]? The answers to this survey provide insight into the most important elements of a successful MBSE project. For best adoption, an MBSE tool must be easy to learn, and understand, meaning the core concepts of MBSE (tool, methodology, language) must be separated and clearly explained. It must have well organized, clear, and efficient code. It must be as useful as possible, providing even the earliest adopters with tools and templates to assist in speedy development of models. These are both consistent realms of improvement for Horizon but not the focus of this work. The survey found that it also must make an effort

to attract new users and developers. Finally, it must make an effort to maintain the community, a similar observation I made when researching activity levels of different open source projects in Section 2.3. Most relevant takeaways to this project are that a project can reap benefits of community adoption and development by providing a forum for community building and troubleshooting development, by being responsive to questions and concerns from the community, and maintaining proper coding, MBSE, and community interaction standards as the community grows.

2.2 Active Model-based System Engineering Projects

To understand Horizon, it is first helpful to gain an understanding the broader field of MBSE. This section is a survey of tools currently used by system engineers and the aerospace community. By discovering common functions and uses, as well as what differentiates the tools, it becomes possible to assess how HSF fits into the field of SE, and future improvements to increase HSF competitiveness and usefulness.

2.2.1 SysML

System Modeling Language (SysML) is a general purpose architecture modeling tool which has become the de facto standard for MBSE. It is an extension of a subset of the Unified Modeling Language, and it was adopted by OMG in 2006, which has continued development since then [2]. The core of SysML is based on block diagrams, which specify system requirements, behavior, structure and parametric diagrams [9]. System structure is used to organize a model and shows relationships of the internal structure in terms of ports, connections, parts, etc. A SysML parametric diagram is designed to describe mathematical equations and are used with constraints which define a basic equation. Parametric diagrams enable engineering models and analysis

to be performed on the model. A requirements diagram depicts how requirements are derived, flow down, and generally relate to one another. Finally behavioral diagrams contain use cases, activity flows and sequencing diagrams as well as a state machine diagram which describe in high level terms the functionality of a system. When combined, these four can serve as a powerful tool to assist in not only visualizing a complex system, but also modeling and providing system analysis.

The creators of SysML have identified four main uses of SysML according to the increasing levels of completeness of model diagrams and system description [2]. The least rigorous and most common use of SysML is referred to as SysML-as-pretty-pictures. This describes when SysML notation is used to build diagrams instead of other common visualization tools like Visio or PowerPoint. This is the least useful for engineering purposes because it rarely is able to produce a simulation or specify system architecture. As stated in the name, it is not used as an engineering tool as much as a picture/ visual diagram producer. According to the creators of the software and website, this is a misuse of SysML [2].

The second is called SysML-as-Model-Simulation. This type of modeling uses both parametric and behavioral diagrams to simulate the dynamics of a system. This type of use is advantageous to engineers as it produces useful data about a system's evolving behavior but often misses important interactions with other dynamic states of a system [2].

A third use case is dubbed 'SysML-as-System-Architecture-Blueprint'. This is an improvement compared to the former use because it includes precise and complete specification of a System Architecture Model (SAM) [2]. A SAM describes multiple views of a system, for example, electrical, mechanical, structural, and how behaviors in each subsystem interact with one another. While this usage describes the system completely, it is not the most rigorous nor the most useful to an engineering project.

The most rigorous and useful application of SysML is 'SysML-as-Executable-System-Architecture', what the creators call a 'quantum improvement' upon the preceding use. This case describes a system where a majority of the behavioral and parametric models are simulatable and potentially executable. This is the highest form of SysML and is the most effective way to implement MBSE best practices with SysML. If a model is executable, it refers to the capability of partially or completely automatically generating system interface and test code.

Of the endless applications and extensions of SysML, notable for this project is an application of an exhaustive search function through the entire state space for a system modeled in SysML. Mehrpouyan et al. used a machine learning algorithm to simplify complex problems into problems which can feasibly be exhaustively searched [10]. The assume-guarantee technique is used to simplify system verification, but required a tremendous amount of work to implement and test this algorithm within a modeled system.

By comparison, the solution space search is the built in analysis tool of HSF, meaning that the verification analysis could be performed quicker, but perhaps less efficiently. Horizon's capabilities fall in between the second and third use cases of SysML. Horizon is a modeling and simulation software so it can easily handle the second use case. While Horizon allows for interactions between subsystems, the modeling capabilities are underdeveloped to use as a complete system architecture model.

SysML has always been built and improved by community development. The community development is different than smaller community developed projects, because the effort was more coordinated by industry leading organizations and experts rather than organic development by passionate developers [2]. As a result, their website contains no encouragement or instructions for new and curious developers to join the effort. Additionally, there is very little public engagement with feature requests and

issues. The current development of the open source project, SysML v2, has separate repositories for its development and release versions. While the release is always initiated by the project steward, the development branch has more than a dozen contributors with considered active and frequent contributors [11]. However, SysML benefits from widespread industry adoption to recruit experienced developers to the project, rather than traditional outreach to users and GitHub community members. This is how SysML has been able to sustain their growth and maintenance for almost 20 years.

2.2.2 AADL/OSATE

OSATE (Open Source AADL Tool Environment) is an open source tool environment used in conjunction with the modeling and simulation language Architecture Analysis & Design Language (AADL). Like SysML and HSF, this software is domain nonspecific, with its intended fields of application ranging from "automotive systems, avionics and space applications, medical devices, and industrial equipment [12]." The capabilities include hazard/fault analysis, system safety/stability/security, performance and flow latency, scheduling, and resource budgeting. Each of these analysis methods produce consistent reports in multiple formats to ease in further analysis. "Models are grouped into separate AADL projects that you can import into your workspace to test and experiment with [12]." Organized libraries of example models assist new users in quickly getting started with modeling, which eases intimidation when adopting a new MBSE framework and tool.

Like HSF and SysML, OSATE is also well suited to MBSE and MDD. The software, which is governed by the AADL models, is meant to be used at each step of the design process for both simulation and communication of system architecture. Producing graphic communications from the models enforces an additional layer of consistency,

which prevents mistakes in describing system architecture and behavior [12]. Additionally, the graphic display can also act as an interface in which users can edit their models. This lowers the barrier of entry for users less comfortable in a coding environment. A key MDD feature of OSATE is its capability to generate code after models are created and verified. Additional tools such as a syntax aware text editor and live updating graphical interface, MATLAB translation, assume/guarantee reasoning and other high-level tools make OSATE/AADL a powerful modeling and simulation tool [12].

OSATE is built on community development too. It provides documentation and instructions on participating in development. However, it doesn't provide an approachable list of first issues or projects to jump into, nor a publicly viewable forum to discuss the state of development [12]. Despite this, OSATE has a handful of core architecture contributors, each with hundreds or thousands of commits and a dozen of supporting volunteers with double digits commits [13]. Because of the high level of involvement of the project architects, release control is done with a manual review process. Therefore, while testing is included in the OSATE repository, it is not automatically executed on each push and does not automatically reject pull requests.

2.2.3 STK

The Systems Tool Kit (STK), formerly Satellite Tool Kit due to its origins as an orbit propagation and visualization tool, is a proprietary physics-based software from Analytical Graphics, Inc. (AGI), now owned by Ansys. The suite of tools allows analysis of ground, sea, air and space systems all contained within the same environment [14]. The backbone of the software remains its geometric modeling and analysis of special relationships. STK supports multi asset scenarios and can model complex interactions over time, as well as optimize a schedule, much like HSF. While

not required, scenarios are capable of comprehensive modeling of every perspective of an asset (electrical, mechanical, structural, communications etc.) similar to general MBSE tools. However, it is more common to perform mostly specialized parametric studies, rather than comprehensively modeling the entire system architecture.

While add-on packages and scripting into the software is possible, STK has been developed as a Commercial off the shelf (COTS) tool, with its native features and functions packed into a polished GUI as a main selling point. These include models of COTS components, 3D animation (useful for presenting concepts to investors), earth terrain modeling, compressible flow analysis, and advanced earth environment models [14]. This saves an organization time, resources, and prevents errors from remodeling these functions, allowing it to instead focus on creating and analyzing their mission. For organizations who can afford the software, which, depending on the license, costs between $200,000 and $10s of millions of dollars, it has been shown to quickly return on investments, improve time to market, and overall increase efficiency for companies [15].

While the capabilities of STK cannot be overstated, they come with significant drawbacks like cost, training, complexity, and rigidity. First, the program can be prohibitively expensive for some organizations with the most stripped back nonacademic license costing an estimated $200,000 [15]. That price tag is enough to deter most CubeSat teams (unless through a university which has licenses), individuals, and small teams or startups. Then, due to the massive expense, combined with the complexity provided by hundreds of functions and applications, a company implementing AGI's solution will often hire a dedicated AGI representative to assist in training, identifying where it can be used, and providing company specific feedback to AGI. This can incur more significant and recurring costs to an organization. It also introduces a limited resource, a single person's time, to a number of teams which may

receive unequal support and training. A final downside to the tool kit is its closed source nature, providing a more limited capacity for customizing native functions, connecting with 3rd-party or legacy software, and general non-transparency. While the mature tool kit comes packed with capabilities, it comes at the cost of ease to learn and implement, a sometimes prohibitive price tag, and a lack of source control over the software.

2.2.4 Horizon Simulation Framework

The Horizon Simulation Framework is freely available modeling and simulation tool developed by students and faculty of Cal Poly San Luis Obispo. It contains a modeling section to describe the characteristics and behavior of subsystems and how data are shared between these subsystems. The simulation portion uses an exhaustive search algorithm to create possible schedules for the modeled system to perform desired tasks. The models are completely independent to the simulation portion. This means that, while the template libraries only exist for aerospace applications, the simulation algorithm can simulate any modeled system in any domain. The output is a list of optimized schedules which the modeled system can perform to achieve target goals. Additionally, the state of each modeled subsystem is returned, which can be given as a set of instructions to a real system to execute the optimized schedule and achieve the system's goals. Horizon is useful at many stages of development, from determining a mission feasibility studies, to component selection, to requirements verification and final scheduling. Further discussion of its capabilities is found in appendix A.1.5.

2.3 Survey of Open Source Aerospace Projects

While the open source movement has been around for decades, the aerospace community has been slower to adopt these processes. This may be due to security concerns, the precedent of expensive and secretive software or simply due to the aerospace engineers not always keeping up with innovations in computer science techniques. Recently, however, there has been greater adoption of open source in existing aerospace projects as well as an increase in the creation of free open source tools by academics and enthusiasts. These provide useful comparisons for the open source release approach of Horizon. Below is a brief survey of open source aerospace projects which have achieved varying levels of success with respect to the cultivation of active community development, a primary goal for most projects going open source.

2.3.1 Open Source, Closed Community Development Projects

Two projects which take a similar open source, yet limited community development approach to aerospace software are JSBSim and MAST. JSBSim is an open source flight dynamics model which, like Horizon, can be run with simple script inputs and has no native graphics. It is primarily a platform which companion programs leverage (e.g., flight simulators, software in the loop autopilot testing, machine learning aircraft control). The source code includes unit testing and regression testing as well as code coverage, which are all run automatically in their CI workflow. The CI workflow remains a focus of their development, as can be seen in the dedicated CI improvement GitHub project. While JSBSim has verified and maintained their software and solidified it as the go to open source flight dynamics modeler, this software is reliant on the significant effort of a handful of dedicated individuals. They do rightfully acknowledge the importance of the symbiotic user-developer relationship for the soft-

ware's "continuing, synergistic improvement" in their 2004 overview paper. Clearly, however, there are improvements that could be made in recruiting developers and better including users in the development process [16].

MAST is another multiphysics engine which specializes in sensitivity enabled finite element analysis. MAST is unit tested but their CI is limited to only a build test on different OS's [17]. The core contributors know to test their code prior to pushing, but they likely do not have the resources to build on multiple OS's. This points to the fundamental difference between JSBSim and MAST's open source philosophy and the community driven development that Horizon strives for, which others have achieved.

While JSBSim and MAST are open source and do have community learning spaces and involvement through feature requests and issue flags, they do not encourage community code contribution to the extent of Horizon, or SU-2 and FlightGear, discussed in the subsequent section. These software operate in a closed development environment, so their resources do not include instructions to fork new feature branches, projects for new developers, nor ways to become involved with the development team. From the lack of community development resources, to the resulting lack of community participation in the software's progress, this barrier between users and developers is clear and undesirable.

2.3.2 Open Source, Open Community Development Projects

FlightGear is an open source flight simulator built on other open source blocks, one relevant block being JSBSim. Flight simulators have many use cases: research, training, commercial, and recreational. This means FlightGear has a large and diverse potential audience, including academics, aviation professionals, and hobbyists. Additionally, the major flight simulators are expensive and closed source, which means

FlightGear fills the gap of an affordable, professional flight simulator. Finally, FlightGear benefits from an enthusiastic flying and flight simulator community. These represent the necessary ingredients for a successful community supported open source project, as discussed in Section 3.1.2. Also discussed in that section is that a project must have both the necessary characteristics and the necessary resources and community encouragement to succeed as a community sustained open source project.

FlightGear is a great example of software with community contribution at its core. There is an extensive developer hub on their wiki with expected pages like in progress and planned projects (of which there are 129 in the 'core development' alone), first issues, and a forum to ask questions [18]. However, FlightGear's wiki goes much further to provide detailed guides for creating new subsystems, render optimization, code cleanup, extending scripting capabilities, and many more. Additionally, there is comprehensive support for non-core developers, those who'd like to add a scene or their favorite airplane. These developers can find step by step guides on every aspect of generating, modifying and troubleshooting aircrafts, animations, scenery, and scripts. They also build their community by interviewing contributors to include in their newsletter, participating in Hackathon events, and a guides to demonstrate FlightGear at expositions [18].

SU-2 is an open source multiphysics framework which was built on a foundation of community driven development. In contrast to JSBSim, SU-2 explicitly states their goal of an open source software engineering strategy. They do this by designing their software as a testbed for numerical methods and CFD experimentation, using high level abstractions for reusability and rapid implementation, and including libraries to reduce user's initial time investment. This also means that, like Horizon, significant features and models can be added without affecting the framework. They also en-

courage global adoption by valuing portability and efficiency: SU-2 is intended to be run on any machine with a C++ compiler and its algorithms are scalable.

Additionally supporting this philosophy is SU-2's extensive developer support. In their manifesting paper, community involvement is mentioned as the first pillar of their philosophy [19]. The project certainly follows through on this philosophy. SU-2 has lively issues and projects tabs for experienced contributors to jump into as well as the same code covenant and a similar standard open source license as Horizon. There is also a wealth of resources on their website like tutorials to make changes to the repository, running containers, code style and review guides, and writing unit tests. It also uses a GitFlow branching model, which is a CD/CI pipeline to merge daily work branches together prior to larger, coordinated releases [19]. While unit tests and regression tests exist, regression prevention is done manually, rather than automated into the version control architecture. Developers are asked to run the regression test suite on their own machine prior to pushing, but eventually the suite is run manually by the reviewing moderator [20]. While this is not as rigorous as preventing push on a failed regression test, the tests are available for anyone to run, which increases transparency and aligns with standard validation and verification practices.

As a result, SU-2 has cultivated an active community of contributors and users which keep the software improving and reaching new users. Additionally, their developer friendly practices have resulted in 92 contributors during its time on GitHub, which represents only half of the software's life. Of these 92 contributors, 16 have over 100 commits and 51 have more than ten [20]. Similarly, FlightGear sees multiple thousands of downloads per week, 23 years after its first release. The number of issue/project tickets tell the story of their active developer base which supports these users. There are a little over a dozen active developers as of April 2022, adding, commenting, and closing tickets [20]. Hundreds of open tickets, thousands of closed

tickets, tens of thousands of comments across these tickets, and dozens more opened each month make it clear the community supported development can work extremely well for this open source project.

Contrast this with similar statistics of JSBSim and MAST, both of which are open source but do not encourage community contribution like SU-2 and FlightGear. JSBSim has a single contributor with a majority of commits (90% of total commits) and lines added or subtracted (80%). After that, only two of 30 contributors have more than ten commits to the project. Clearly, this project is driven by an individual with a few occasionally supporting contributors. With MAST, the contrast is stronger. There are only 4 developers with commits, of whom the main architect designer is responsible for 76% commits and 74% lines of code added or deleted. One unfortunate product of this is that, as of writing this in April 2022, the MAST architecture has not seen any commits in nearly two years, and the latest release is from January 2020.

This information is useful in directing how Horizon should grow. As previously mentioned, Horizon would benefit from more development from both dedicated developers who can add new features and expand the architecture and users who can add to the program by adding subsystem instances to libraries, suggest features and point out bugs. To get developers, Horizon can rely on recruitment of master's students or can reach out to the developer community for support. Clearly, the latter method has worked for SU-2 and FlightGear to maintain consistent development over very long periods. Therefore, Horizon will pursue their method of community driven development, building resources and a community slowly over time.

2.4 Thesis Statement and Motivation

The development team of Horizon have decided that the next logical step in the software's progression is to go from closed development, free-to-use, as-is software to a collaborative open source project. The primary goals of this paper are achieved by addressing the lack of testing and validation, integration pipeline, and other necessary supporting documents expected for a prosperous, collaborative project.

2.4.1 Motivation

Horizon has been freely available online since 2015. The code is presented as an 'as-is' software, meaning the user is given no guarantee of the software's accuracy, usability or state of completion. For Horizon, this has meant that there is no development version separate from release version of the software, so a user may download the software with a half-finished feature or even with broken core functionality during adjustments to the architecture. The work presented in this thesis aims to move Horizon away from a perpetually in-development phase to a functional product with certain guaranteed functionality, even during continuous development. The Horizon team would also like to see an acceleration in development for Horizon, including more simulation features, modeling features, templates, and external interfaces. Releasing Horizon as an open-source modeling and simulation platform, with an emphasis on community driven development is the proposed solution. However, the software was deemed unfit for release because Horizon lacked the testing to verify its functionality, and it lacked a system to maintain the integrity of the code after its release to the community. Additionally, the software has disorganized or missing developer support resources.

2.4.2 Current Barriers

A primary barrier to a formal release was that no one had ever verified Horizon. The individual methods had not been tested, nor the larger program modules. Therefore, the output of the program has never been systematically compared to expected results calculated outside the Horizon program, and no conclusions should be drawn from its output exclusively. But it is the hope of this development team that in the future this software is used by more than just this team. However, it is absurd to expect an organization to choose an untested software for two reasons. First, as stated above, it is inadvisable to draw a conclusion from a code base which has never been verified. Secondly, the fact that the developers did not take the time to test and provide users with proof that it works, reflects poorly on the quality of the software. The result is a software, intended to be used by the public, being neglected by the public. To convince users that the software is high quality, they need assurance that the product works as intended and remains working as intended. This is a central challenge my thesis works addresses.

Another barrier is the small dedicated moderation team, often consisting of a single person to deal with an entire open source project. In the desired event of medium to large scale adoption and active contribution from a community, it would be impossible to manually address each commit, run tests, and maintain the code quality. Without a test suite to compare versions of code to, the maintenance is impossible to perform quickly and effectively. The result would be either a backlog of commits to be manually checked for bugs, or a buggy architecture.

The final barrier is the lack of organized documentation that developers expect when considering joining a community project. When a new user arrives at the GitHub repository, there is little direction as to where users should go for more information.

Users looking for a first project to contribute, more information about HSF, or how to download and get started are all met with the same generic landing page. There is also no open source license, contribution guidelines, nor code of conduct. This might steer new users away from the disorganized repository and result in a lack of community development.

2.4.3 Automation for Maintenance

The solution to the problem of unverified code is to simply verify it. For a static piece of software, a single verification is enough. However, Horizon is dynamic with new features always in the pipeline or wish list. So, because code and new features from unknown community contributors is encouraged, verification of the core architecture must happen every time the code is changed. This can be done by manually testing then accepting or denying pull requests or it can be automated to run each time a push or pull request is initiated. Automatic testing is well suited for projects with few or no developers dedicated to maintenance. Any user curious about the current state of the repository can easily glance at the test results of the latest CI run and gain confidence that the main scheduling algorithm is verified and properly functioning.

The second barrier of quality and speed of handling pull requests is also addressed by the CI Pipeline. As previously discussed, the owner and long-time sole maintainer of the Horizon project, Dr. Eric Mehiel, has little time or interest sifting through change-logs and testing the new code himself. Instead with the CI pipeline, if a pull request fails a test in the pipeline, Dr. Mehiel or other moderators can easily see what tests are broken and give feedback and/or reject the request. This way, the moderation team can more quickly and effectively approve or reject pull requests based on the test report generated with each commit.

2.4.4 Thesis Statement

A continuous integration pipeline with automated testing was identified as the solution to the open source barrier of lacking testing and maintenance. First, to verify Horizon, unit and integration testing is needed for each referenced method and class in the scheduling segment. These tests enable the implementation of a continuous integration pipeline, which executes the tests suite for each commit to the repository. To address the final issue of an unwelcome homepage with missing documentation, the repository must be cleaned and organized. The file tree must be organized, a README.md should direct new users from the landing page, the previous OSS documentation should be assembled, desired features and new library ideas should populate the projects tab, and all Horizon resources must be found in the wiki.

Chapter 3

AN OPEN SOURCE HORIZON SIMULATION FRAMEWORK

This chapter focuses on Horizon and its open source future. Open source software is explained as well as the reasons to release and contribute to this type software. The main goal of an open source software is to be used by a wider audience and receive improvements through community participation. To attract community development a project must satisfy certain needs for the developer community, maintain logical organization and a respectable appearance, and contain adequate resources for all types of community members who may stumble upon the project. To identify what is needed for Horizon to succeed as an open source project, this chapter pulls heavily from experts on open source and GitHub, the largest host of open source software.

3.1 Communities of OSS

Open source software (OSS) is broadly defined as software where the user can use, modify, and redistribute the software without permission or compensation of the author [21]. Practically, when a software chooses to go open source, they invite the public to scrutinize the source code and improve the software in collaboration with the original team. Specific requirements for OSS have been defined by watchdog organizations such as the Open Source Initiative (OSI) and the Free Software Foundation (FSF). According to GitHub [22], there are three main reasons to open software to the public: collaboration, tweaking/remixing, and transparency. While transparency is not currently of concern to the HSF team, collaboration and essentially outsourcing the creation of new models, libraries, and add-on features, is.

3.1.1 Doubts about Open Source Software

Many computer science scholars have doubts about the promises of OSS however, especially when considering applications with "stringent requirements for certain attributes such as security, reliability, fault tolerance, human safety, and survivability, all in the face of a wide range of realistic adversities-including hardware malfunctions, software glitches, inadvertent human actions, massive coordinated attacks, and acts of God [21]." This application accurately describes the challenge of writing aerospace software which claims to be able to design, test, verify, and control systems which can have costs reaching in the billions of dollars. The question posed by Peter Neumann, a leading academic on the intersection of society, policy, security and software, is, "is open source software better than closed source, proprietary software? [21]" His answer is that it is not intrinsically better, but it has the potential to be.

> "The potential benefits of nonproprietary nonclosed-source software also include the ability to more easily carry out open peer reviews, add new functionality either locally or to the mainline products, identify flaws, and fix them rapidly-for example, through collaborative efforts involving people irrespective of their geographical locations and corporate allegiances."
> -Neumann [21]

According to Neumann, the main disadvantage of for a company choosing to use open source software compared to a closed system mentioned in the analysis is "increased opportunities for evil-doers to discover flaws that can be exploited and to insert trap doors and Trojan horses into the code [21]." However, for a low profile software like Horizon, it is unlikely that anyone would be motivated enough to carry out this attack. A much more likely disadvantage with allowing community development is poorly written or buggy code introduced which breaks core functionality. Therefore,

security concerns are left to the community and developers must passively or actively patrol the code for malicious code. Additionally, security is also partially deferred to the judgment of the user, through allowing scrutiny of the open source code prior to adopting the software.

What is not considered in this analysis by Neumann, but more specifically relevant to HSF, is OSS's particularly well suited benefits to software projects with a lack of developers in the software's managing organization. The majority of HSF has been developed by graduate students at Cal Poly with sometimes years between students working on the project, as was the case from 2012-2016 when the software had no development. Therefore, an significant disadvantage of this framework remaining closed source is the slow, sometimes halted development of new features, models, and implementations. While general advantages of OSS have been discussed in section 1.2.2, the current primary goal for Horizon's team, developing new functionality and instances of simulations without the need to recruit new graduate students, is satisfied with OS development, while the issue of security and safety intrusions are not considered a likely outcome, and the issue of bad code introduced into the framework is addressable.

3.1.2 What makes software a good candidate for Open Source

The success of an open source release can often be predicted by looking at the intersection of the project and it's developer community. A project must scratch an itch for people who code; a large potential user base is not enough for an OSS project to succeed [21]. Similarly, the project must attract a large or dedicated developer community to remain actively in development. Another key component in many successful open source products like Linux is that the product serves as an alternative to expensive corporate software [21]. A software which fits these criteria may show

potential to intrigue an open source community, but it must have desirable technical characteristics to back it up.

Success also lies in the characteristics of the software which can also be broken down into design and tools. A poorly designed architecture or a massive monolith can make development too daunting for potential coders. Development tools (revision control, bug reporting databases, etc.) are important components for an open source project to have a reasonable chance of success [21]. Much care was taken during the recode of HSF to make the architecture more logical and organized. Horizon also uses tools like GitHub for version and regression control and takes advantage of the expansive developer toolkit included in Visual Studio Community Edition. Finally, success for a closed source project is often ultimately determined by the team who chooses to cancel or continue development. However, it is common for an open source project to appear failed at first to later find popularity and become a success.

An open source Horizon Simulation Framework will be successful if it has an established user base, creating and sharing models, comparing simulations, and helping others learn. Equally important is that a portion of these users become developers, who create, maintain, and debug new features, templates, and add-ons. Unlike most software, a majority of users will interact with HSF through code, so the lines between developers and users is blurred. Users cannot simply open the program and begin building models without a basic understanding of scripting and the framework. Similarly, it is unlikely there will be developers who are not using the software to run their own simulation. This addresses the 'people' problem for successful open source software projects, because all interested users result in likely developers of the project.

Horizon also addresses a need for a free alternative system modeling and simulation tool, with powerful design and verification capabilities. HSF can be considered

an open source alternative to STK, FreeFlyer, and MATLAB. It should also attract aerospace engineers who code and are looking to perform rapid mission analysis without the massive learning curve associated with other MBSE solutions like SysML and OSATE. A key advantage to HSF as an open source MBSE tool is that it avoids being too abstract and thus unappealing to prospective developers, while being broad enough to appeal to a wide multidisciplinary pool of users. The team also considers the project highly modular. Because the framework consists of the algorithm and supporting libraries and features, further development of the software is achieved with the addition of relatively simple functions and subsystem models, a task easily achievable for an individual. It is clear that Horizon's concept and architecture are well suited for a future in open source, but there is more to open source success than a well suited project.

3.2 Necessary steps prior to releasing open source software

Most of the aforementioned criteria for a successful open source software candidate relate to properties about the core concepts of a project. There are, however, still critical steps that a team with an already realized, well suited OSS candidate must take to prepare for an open source release. A crucial step for sharing any code base, open or closed source, is providing proper documentation. Documentation shows what it's for, how to use, and other organizational, legal and miscellaneous information the developers need to share with anyone downloading it.

3.2.1 Minimum Required Documents

GitHub identifies four documents that should be included in every OSS: open source license, README, contributing guidelines, and a code of conduct [22]. First, the

open source license can be drafted by a legal team or lawyer, but most small teams use a standard template from either OSI or FSF. A README is the first thing a user should encounter when discovering or open after downloading the software, which details what the project does, why it is useful, how to get started, and where to get help if needed. Contributing guidelines tell potential developers how they can participate and the expectations of participation. For a young project, this may also be a good place to suggest 'first issues' or 'first library functions' which can draw in new developers with simple yet important development ideas. Finally, a code of conduct describes the expectations for all interactions between users, developers and maintainers, and is also commonly adapted from templates from OSI and FSF. Often, the code of conduct and the contribution guidelines are the same document.

3.2.2 Organization and Appearance

In addition to the standard documentation included in most OSS releases, there are other actions a team may take to increase their chances of success. While clear, commented, and well organized base code and supporting documentation may sound obvious for a software team whose goal is to attract unfamiliar users and developers, GitHub's 2017 Open Source Survey showed that incomplete or confusing documentation is the biggest issue for open source project users [22]. Thus, it is key to fill in documentation gaps, and strictly enforce guidelines when reviewing a project prior to release.

3.2.3 Community Growth and Retention

Another logical step is to actively advertise and increase awareness of the tool after its release. This can be achieved with online outreach in places with lots of engineers and

developers such as Reddit, StackExchange and similar coding communities, LinkedIn, Facebook and direct to small teams who may benefit from the software [22]. Additionally, offline outreach can be powerful in growing a community as well. For HSF this could include submitting conference papers about HSF, partnering with Cal Poly clubs and teams, or reaching out to local companies and alumni.

GitHub recommends that OSS communities have a specific place for public congregations where users, developers, and maintainers can all interact, ask questions, and generally foster a community (e.g., Twitter/Facebook page, dedicated software website with a forum, Git forum) [22]. A public congregation can also help by enabling community members to answer questions before a core member is able to respond, and for maintainers to avoid answering redundant questions. A third function of a public online congregation is to show off interest and active engagement, which increases visibility on the web and will make potential developers and users more likely to try it out.

3.2.4 Continuous Integration

Continuous integration (CI) is a great practice for any project, but provides especially useful benefits for OS projects. Because anyone can contribute but few maintain OS projects, they are particularly vulnerable to poor quality code and poor and/or laborious quality control. CI is effective at addressing these two issues.

Continuous integration is often attributed to Martin Fowler, an expert in software development specializing in the process of software design. His succinct definition is:

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates

> at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly [23]. - Martin Fowler

Fowler states that the difference between projects which experience integration as an arduous and unpredictable process and those when treat integration as a non-event which can be done in minutes, is not due to expensive or complex tools. Rather, the result is due to the simple practice of the whole team integrating frequently. Continuous integration with source control software follows these steps. First the mainline source code is forked to a development machine where it is changed (e.g., new feature, bug fix, etc.) Then tests are made for this new change. CI relies on comprehensive testing of the units and functions which are automated into the software, often with assistance of a test framework. A recommended but not automated or enforced step is that the developer then runs the test suite on the local copy to work out any obvious bugs. The code is then merged with the mainline code, a step which is only necessary when the mainline code is modified while this copy of the code is checked out and changed. Merging is commonly required for active projects with CI because the point of CI is that forked copies are frequently integrated and thus it is likely that someone else has committed changes during any given feature development. Once the code is properly synchronized with the mainline of code, the new merged copy can be committed to the mainline. The commit triggers a series of automated actions. The automated actions often include a build of the software as well as running some test suite. Once the code successfully builds and passes automated tests, the code is either automatically accepted as the new main or is ready for one last manual acceptance, completing the integration process.

While this practice can be implemented by a change in behavior of a team (simply requiring everyone to integrate more frequently), it is always useful to use a CI server. A CI server is used to build the program, run tests, store version history, and deploy software and can be automated to run different workflows based on desired triggers. Examples of CI operations include running tests for any commit made to the repository, and incrementing the version number with each release on the software's web page. When implemented well with good CI practices, the resulting software should always be of quality, have few bugs, be up to date, be stable, and function according to the design.

For open source projects, CI provides a valuable service in automatically testing source code. Since Horizon's maintainers have limited time to dedicate to combing new commits for bugs, its automated testing can significantly speed up the code review, and thus the integration and acceptance process. This is both beneficial to the developer who receives immediate feedback from the automated build and test on whether the code is likely to be accepted, but also to the reviewer who does not have to manually run tests and can approve code with a cursory review of the edited files.

3.3 Making CI Effective

The success of implementation of CI is dependent on many key practices. While the concept is relatively simple, there is much more to setting up a smooth CI pipeline than adding software to the pipeline. Fowler lays out 11 common sense practices that he's seen help create effective CI pipelines. These practices will be used to inform the approach for setting up Horizon's CI pipeline.

3.3.1 Best Practices

First is that a single repository is maintained. Source code management tools are integral to keeping track of the many files and versions. Second is automating the build, which means that "anyone should be able to bring in a virgin machine, check the sources out of the repository, issue a single command, and have a running system on their machine [23]." Third, the build should self-test. This is most often with the help of a testing framework which works with CI servers to execute different combinations of tests. Automated testing is critical to CI's effectiveness in finding bugs. Fourth is that commits are made frequently. Fowler advocates for committing every day, which encourages developers to break code into small chunks, facilitates communication between developers, minimizes the number of lines of code which must be checked if a test is failed, and minimizes serious conflicts by forcing any issue to be addressed immediately. Fifth, every commit should build the mainline on an integration machine, best done on the CI server. Fowler advocates that no-one should leave until receiving a notification that the build has succeeded because it enables the next best practice. Sixth, fix broken builds immediately. A significant benefit of CI is that it facilitates the quick identification of bugs. However, identifying a bug quickly is only useful insofar that it is quickly removed. This is good practice because, somewhat obviously, bugs are bad to exist in the mainline code, but also because code is developed in such small chunks, the possible location of this bug is narrow if it is addressed as soon as it's found. Seventh is that the build is fast so the pipeline is not bogged down by overlapping commits. Fowler generally recommends keeping the pipeline run-time under 10 minutes. Eighth is testing in a clone (or as close a mimic as possible) of the production requirement to reduce risk associated with testing the software in a different environment than its intended use. Ninth and tenth is that it is easy for anyone to get the latest executable and that everyone can see the history of

changes and test results. Finally, the last practice to ensure an effective CI pipeline is extending it to automatically deploy to users. If scripts are already written to deploy the software in different environments (to build on the CI server), it should be easy to create a similar script which deploys the software to production. These guidelines are sufficient for most projects setting up a CI pipeline.

3.3.2 Additional CI Efficiency

There are studies on advanced implementations of CI that address problems which arise with scaling up software projects. These include innovations like selection and prioritization techniques used at Google, which led to cost effectiveness improvements [24], and a technique which applies fault-based and risk-based test selection optimized for low run-time, which achieved improvement on time-efficiency with respect to industry practice [25]. However, this thesis focuses only on the basic guidelines for implementing effective CI, described by Fowler. This is because time and cost effectiveness are not primary drivers for this open source, volunteer and student supported project and have not hampered implementation of CI. Low cadence of commits for Horizon means that CI builds rarely, if ever, overlap and lead to conflicts. Similarly, Horizon has a small enough code base and simple enough tests that a pipeline can run within the guideline of 10 minutes without any optimization.

3.3.3 Results from a CI Implementation Survey

An analysis on the quality and productivity outcomes relating to continuous integration in GitHub by Vasilescu and Yu et al. found that projects who can integrate more outside contributions see an increase in productivity without an observable diminishing of code quality [26]. The analysis considered projects with a significant

number of pull requests (< 200), to focus on projects which should benefit the most from CI. They analyzed projects which had a significant portion ($< 25\%$) of pull requests prior to implementing CI so they could analyze the quality and productivity before and after for each project. They found that CI use had a significant positive effect on the number of merged pull requests and negative effect on rejected pull requests from core developers. The analysis found no increase in accepted requests from external contributors, but it did find a decrease in rejected requests, suggesting that CI provides immediate feedback which lets these contributors quickly fix failing tests. Additionally, the authors found an increase in pull requests handled after CI adoption, meaning core members are able to more efficiently handle pull requests. Importantly, this increase in productivity comes at no cost of user experienced quality. User reported bug reports did not increase significantly alongside the increase in volume of code managed. However, bug reports from core developers increased by nearly 50%, meaning that team members are better at identifying and fixing bugs. Other interesting findings of the study were that the older and more popular a project is, the fewer bug reports and pull requests are submitted by core developers, suggesting that maturity correlates with quality. It may also suggest that older, more popular projects rely more heavily on user bug reports. Overall, they concluded that adopting CI has clear benefits: more pull requests processed, more code being accepted and merged and at no cost to code quality [26].

Chapter 4

METHODOLOGY

To improve Horizon to a tested and open released state, the type of testing, framework to test with, and the scope of testing must be defined. Testing takes several forms with each serving a specific purpose in the software development cycle. Contrast Horizon's release and audience with the roll-out of a new iOS application and it is clear that testing and release protocol differs greatly between software types. This chapter focuses on defining the specific testing and documents needed to ready Horizon for its open source release.

4.1 Testing

Testing is an important step prior to releasing software because it helps eliminate bugs before they have a chance to taint the reputation or provide users with harmful information. Testing mostly falls into four categories: unit testing, integration testing, system or functional testing, and acceptance or user testing. Unit and integration testing are primarily performed alongside development, whereas system testing and acceptance testing focus on the users experience with the finished software. My work focused on unit and integration testing and some system testing. The user experience side of Horizon is underdeveloped, thus requiring more development before user testing is useful. Regression testing, the final relevant tests for Horizon, is performed by executing some or all of the previously mentioned tests with each update of the software. These tests ensure that existing functionality is not broken with new changes.

4.1.1 Validating Units

Testing software involves comparing data that the program or unit produces to the expected result. There must be great care in determining the correct solution to give the test to compare to the output. If a bad value is given to the test, then the test is useless because it tells the programmer nothing about the correctness of the subject of the test. The expected result should always be independently validated, when possible, by computing the answer using something outside the program. Solutions may be obvious, requiring no effort except a glancing thought, they may require manual calculation, or they may be complex enough to be best solved by another previously verified program. I will elaborate these three methods using examples of functions I verified for this project.

The `Accepts()` function in the `SingleConstraint` class returns a true or false based on whether the passed in value is compatible with the constraint. For example, a constraint could be the angular acceleration provided by the reaction wheels can never be greater than 1. To write a test for this case, it is easy to identify the desired output based on any input. Negative numbers and numbers between 0 and 1 should return true, 1 and any greater number should return false. Another example is the `integrate()` function in the utilities project. Simple results can be verified by hand or done with numerical programs like MATLAB. Finally, are the complex functions of Horizon like the 3-D vector geometry problems. Functions like has line of sight (`hasLOS()`), which returns whether an asset has an unobstructed view of a point, and `castShadow()` which returns whether or not an asset is in shadow (or is in the penumbra). These functions are not built into numerical programs like MATLAB or Wolfram, and thus must have more complicated verification. However, these problems have been solved for decades and have publicly available solutions. For these functions, I found an implemented solution from a trustworthy source. Both of

these functions I verified with Howard Curtis's freely available MATLAB code, who literally wrote the book on orbital mechanics. These methods provided the integration and unit tests with the hard coded 'true' values to compare the actual output of the functions.

4.1.2 Unit Testing

Software is developed in parts or units and each unit is expected to function in a defined manner. A unit may be a method, module, or object. The purpose of a unit test is to ensure that the unit satisfies this function and that its implementation is consistent with the intended design architecture [27]. In Fig. 4.1, a unit test is annotated to show the anatomy of the test. The best practices are discussed later in this chapter, but from a broad lens a unit test should be as simple as possible to test all reasonable uses of a single unit of code. Unit tests are great for systematically testing a bit of code which provides a quick check to make sure each part of the program is doing its job correctly.

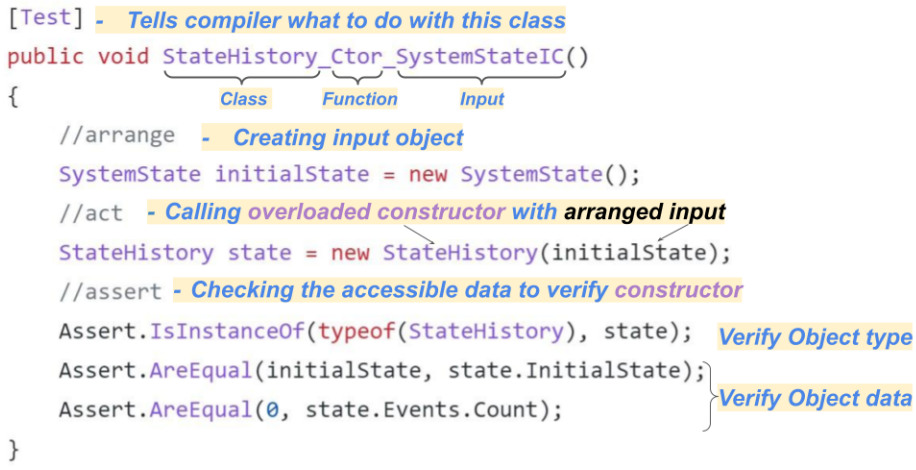

Figure 4.1: Annotated unit test

4.1.3 Integration Testing

Integration testing logically combines units into groups to test higher functionality. This testing can be done with a big bang approach, where all units are integrated in a single test, or in incremental tests where modules are integrated one at a time until all modules are integrated and tested [28]. In Horizon's architecture, there are many higher function modules which call multiple lower dependent functions and are dependent on possibly dozens of objects. When writing unit tests for these, it is very difficult to completely isolate the module to perform a textbook unit test. For a high level method to be completely isolated, writing numerous mock methods using an add-on like Mockito, in addition to dozens of mock objects are required. This may further separate the test behavior from the real use behavior. Instead, I opted for a bottom up integration/unit test hybrid, shown in Fig. 4.2. Bottom up integration testing is where the lowest units are integrated first and the program builds from this until the highest functions are tested with the fully integrated suite below it. One benefit of this approach is that when a test fails it is easier to find the root cause, because the simplest, most isolated tests run first. For these integration/unit test hybrids, I use as many mock objects and as few functions as possible during setup, but I allow the function to call use the real dependent functions. In some cases, when arranging the necessary objects during the setup stage, I opt to call methods which create objects with inaccessible constructors or which involve a cascade of dependent methods to create, simplifying testing significantly. This is consistent with a testing philosophy that says tests should mimic how the code actually runs as closely as possible.

In the test shown in Fig. 4.3 the subsystems objects and relationships are first arranged by the `LoadSubsystems()`. There is a try-catch which alerts the user if `LoadSubsystems()` is the reason for the test failing. A primary disadvantage of using

dependent functions which detracts from a test's usefulness is not knowing which element of the test caused the failure. Then the method which is being tested can be called, in this case the `AccessSub` constructor. That constructor uses another external method, `getSubNameFromXML()`. While this function happens to be tested prior to the `AccessSub` constructor, as would be best practice, this is not always the case. During a CI run, a nonfunctional `getSubNameFromXML()` would fail its own unit test prior to executing the `AccessSub` test and, therefore, eliminates confusion as to which function is not functioning. The tests are generally run in a bottom up (unit to full program) order for this reason, but this guideline is broken regularly because of the complexity and inter-dependency of the code. Therefore, it is a very useful step for developers to run the test suite on their machine because all the tests are run, regardless of any tests failing.

The need for hybrid unit/integration tests is primarily the result of highly complex, coupled code. Most units require many objects and relationships and also call functions of one or more other units of code. For methods like these, a refactoring to decouple methods is a better long term solution. If refactored properly, the dependent methods (methods called by method being tested) would instead, be called first and the objects they create are passed into the subject method. So, when testing, mock objects could be constructed and the method could be tested directly. This is one reason it is a best practice to write tests in parallel with the program code. Writing tests early provides immediate feedback: if a bit of code is annoying or complicated to test, it is much easier to change course while writing the code. Coupling is occasionally unavoidable so it is not necessarily a bad practice. However uncoupled code is easier to unit test then integrate, rather than Horizon's case where coupled code forces certain functions to integrate lower units.

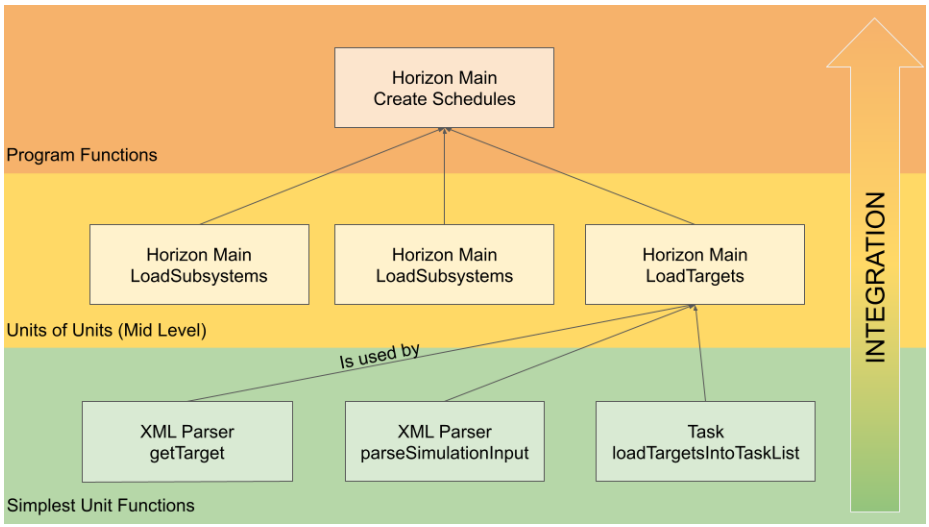

Figure 4.2: Integration testing flowchart

Figure 4.3: Annotated integration test

Writing tests from the start is good practice for many more reasons. For one, the developers who wrote the original code will have the best understanding of the unit and the function it provides to its group of related modules. This saves a quality assurance team or a new developer significant time spent understanding the unit and its purpose. It also saves time and money by catching mistakes and bugs earlier in the

design process. This practice was unfortunately not adopted during the re-code and subsequent development of Horizon, and it is the focus of my thesis project instead.

Horizon was written and is maintained by aerospace engineers with a passion for coding, not by computer scientists. From the start the goal has been to create software that is highly adaptable without changing the core architecture, and which follows coding principles like consistent naming, a logical and consistently applied design architecture, and abstraction. However, the authors are trained more in MBSE and that is reflected in its consistent adherence to MBSE guiding principles, often over coding principles. This priority has resulted in code which is not necessarily optimal from a computer science perspective, specifically to this paper, code which is occasionally difficult to distinctly write both unit and integration tests. This testing campaign is the first step to introducing better coding practices like testing driven development, which improves the efficiency of writing tests and effectiveness of the tests for catching bugs.

4.1.4 Regression Testing

Regression happens to a program when a feature is broken as a result of a software update. Regression testing seeks to prevent this by re-running core functional tests after any change to ensure proper function throughout the software. This can be done in three ways: testing all, selective testing, or prioritization testing. Regression testing where all tests are re-run can be expensive and time consuming when dealing with a large program. However, when a test suite takes a matter of seconds or minutes to execute, running all tests becomes a simple, comprehensive, and feasible solution for preventing regression. Selective testing limits the scope to tests which are most relevant to the affected code or testing the functionality that is most sensitive to code changes. By eliminating many test cases, time and resources are saved.

The prioritization technique aims to reduce the time the test suite takes to detect a fault. Priority can be assigned in several ways such as statement coverage, function coverage, fault-exposing-potential, or most syntactic differences. Prioritization can be combined with selective testing for a hybrid scheme which prioritizes the subset of selected tests. Because an estimated half of software costs go towards maintenance, it is crucial to use and optimize techniques like regression testing [29]. Horizon's entire test suite can execute in under a minute. Therefore, it was decided that all tests should be run for each change to the repository to prevent regression.

4.2 Testing Environment

The testing environment is composed of the integrated development environment (IDE) which is used to organize, view, write and track testing, and the testing framework, a which provides a logical framework to support writing and executing the tests. Below is a brief comparison of testing frameworks and discussion of some helpful tools included in the Visual Studio IDE.

4.2.1 Test Framework

To test a program, a developer could write user code which test various units of the code with assertions, exception handling and other feedback mechanisms to signal failure. Naturally, to better diagnose which tests failed, an output display might be added. Finally, to run selected tests, see summary reports, and generally improve the testing and debugging experience, a GUI can be added to interact with the test suite. Most developers forgo this labor, instead choosing to use the numerous available testing frameworks, which include all of these features. Testing Frameworks eliminate barriers for developers to adopt unit testing.

4.2.2 Survey of Frameworks

The three frameworks that are used most commonly are MSTest, XUnit, and NUnit. They all perform the exact same task of testing with slightly different flavors. MSTest and NUnit are built similarly and most of their attributes and asserts are analogous. Users have reported memory leak problems with MSTest in addition to the issue Horizon faced which was with tests being run in random orders combined with variables which weren't deleted after a test completed. This caused tests to fail because they were run out of order rather than for legitimate errors in the code. One final difference is that MSTest is controlled and developed by Microsoft, whereas XUnit and NUnit are both open source. While I've discussed some downsides of using open source software, they can be ignored in this case because both open source frameworks enjoy regular updates and support, and they have enjoyed widespread adoption for commercial use.

XUnit is leaner compared to NUnit and MSTest, meaning it has fewer asserts and attributes than the alternatives. The lean nature of XUnit testing effectively enforces the design philosophy of the authors; so rather than providing multiple ways to design a test (for example using one or many asserts per test), XUnit informs the design according to the XUnit philosophy (only allowing a test method to have a single assert).

NUnit is the oldest and one of the most popular testing frameworks for .NET. Other than giving a certain sense of security in the continuation of the project, it also means that there are a lot of third-party resources, guides, and tutorials available for NUnit. Overall, the differences between the frameworks are quite subtle, and the only consensus in online communities is that they will all work.

4.2.3 Switch from MSTest to NUnit

Because the testing was in nascent stages at the start of this project, it was a great opportunity to re-assess the testing framework that Horizon uses. Initially, the team decided to stick with MSTest because there were already a few tests, and there is no significant difference between frameworks. However, when implementing CD/CI through GitHub's Actions, the test files were either not found in the file tree, or not executed by Git's servers. Additionally there was the previously stated issue where variables would not clear between tests and it was difficult to discern why tests were failing in the pipeline. Limited documented instances of GitHub Actions with MSTest, in contrast with NUnit, and the ease of switching between the frameworks influenced the decision to try NUnit as the testing framework. Additionally, NUnit enjoys slightly wider adoption, and has more community activity with its users and the original developers. Because MSTest and NUnit are so similar, "Find and replace" was used to switch attribute keywords, and nearly every assert statement worked in both frameworks. The only other required work was to add the NUnit NuGet packages to each project and fix a single bug related to relative file paths. Using the NUnit test framework, the tests ran flawlessly on GitHub's remote servers, meaning the CI pipeline was complete.

4.2.4 Visual Studio IDE

Visual Studio is a powerful IDE built by Microsoft. C# is well integrated into Visual Studio, a driving factor in the decision to switch Horizon to C#. Visual Studio has features like IntelliSense, an autocompletion feature, which shows users suggestions of accessible variables, objects, methods, classes, and interfaces. This is extremely helpful to those without a coding background and developers who are unfamiliar with

C# or the particular software architecture. Another helpful feature of Visual Studio is the built-in test matrix GUI, shown in Fig. 4.4. Any method with the `[Test]` attribute will be collected by the matrix and organized in its respective assemblies. When a test is run, the test matrix updates the outcome of the test. Passing, failing, and inconclusive tests are marked with a green check, red 'X', and a blue '!' respectively. Methods which are called during any tests also feature this marker below the method header indicating whether all tests which call that method have passed. The user can hover over to see all the tests and their outcomes and jump straight to a failing test to investigate the failure. Visual Studio's test matrix and other integrated testing functions are compatible with their own MSTest framework and 3rd party frameworks.

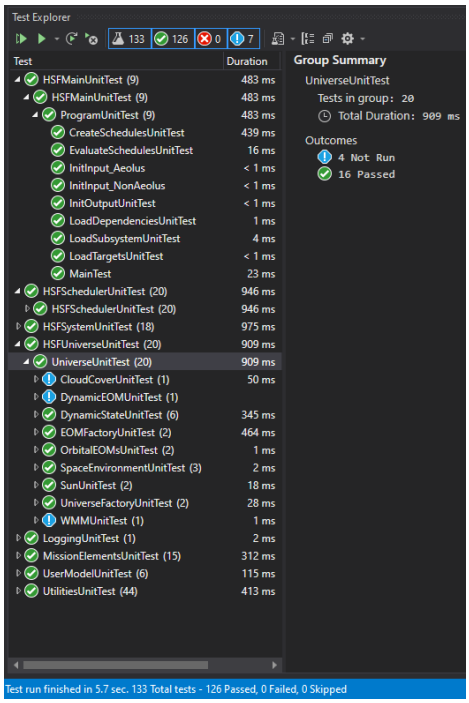

Figure 4.4: Built in test matrix in Visual Studio Community 2019

The matrix is not only useful for quantifying the number of tests, but also as a way to keep track of testing progress. Empty tests with an `assert.Inconclusive()` can be used to populate a test matrix, giving the developer an idea of the number of tests and

therefore work ahead of them. A quick glance at the test matrix shows which tests are to be written, marked with the blue exclamation mark. As testing continues, the number of inconclusive tests acts as a countdown until all planned tests are written, and no inconclusive tests remain. The test matrix's integration with Visual Studio removes many barriers to implement testing, and it is an incredibly helpful visual interface for developers.

4.3 Best Practices

Unit testing aims to encourage better code design and reduce development timelines by reducing time spent functional testing, fixing regression defects, debugging, and rolling out. However, poorly designed tests can seriously hinder development. Best practices should be followed to avoid common mistakes which lead to ineffective testing.

4.3.1 Fast

Larger projects may have thousands of unit tests. In the best-case scenarios, slow tests can be an annoyance to developers who must wait during long test runs. In worst case scenarios, it can clog a CD/CI pipeline which can't complete test runs in between pull requests. Therefore, it is expected that each unit test takes as little time as possible to run, on the order of milliseconds.

4.3.2 Self-checking and Repeatable

Unit tests should be able to automatically detect if it has passed or failed, that is, without human interaction [27]. The assert function is included in all test frameworks

for this purpose. Test runs should also provide consistent results between runs if nothing is changed. In unit testing, the use of objects or methods which may change their outputs over time should be avoided (e.g., Current time, random generators, non-local databases).

4.3.3 Uncoupling Code

Unit testing alongside development encourages less coupled code. Because highly coupled code is, by its definition, hard to isolate, testing involves constructing many objects and methods which are not the subject of the test. This is bad practice because it makes locating the root cause of failure more difficult, and it makes tests run longer than necessary. In the arrangement of a test for a new method, a developer may notice that there are many objects and method calls which don't provide relevant information to the function or object being tested. This feedback naturally decouples code because it would be more difficult to test otherwise, resulting in simpler, more easily understood, and more easily tested code [27].

4.3.4 Naming

Organization is instrumental in delivering the benefits of testing. Tests are only useful if they are easily understood and assist in rooting out bugs. Test naming is an often overlooked but instrumental organizational tool. Tests should have specific names which indicate which method or object is used and the specific aspect being tested. So rather than `Object_Test1()`, a developer should use a more specific name like `Object_function_ErrorExpected()` or `Object_Ctor_typeOverload()`. Good names describe the behavior of each test, eliminating the need to dissect the code itself in the case of a failing test [27].

4.3.5 Arrangement

Consistent code arrangement is an effective organizational tool which increases readability of test code. Tests are commonly conducted in three steps: arrange, act, and assert. The 'arrange' step gathers the necessary structures and objects to perform the action. The 'act' step is the call to the method or constructor that is being tested. After the action, the 'assert' step compares the resulting objects to expected values or objects, using assert statements. Separating these sections with comment headers helps communicate the purpose and structure of a test at a glance, and prevents mixing actions with assertions, which hinders readability [27].

4.3.6 Avoid Logic

Logical statements increase the likelihood of an error and make code less readable. Logic statements like if, while, for, foreach, switch, etc. should be avoided. If tests are complicated to understand or were difficult to write a passing test because of tricky logic, a developer is less likely to trust the tests. Developers should take test results seriously and have confidence that a failed test is the result of faulty code. Tests which aren't trusted are useless to developers [27].

4.3.7 Test Private Methods by Calling Public Methods

Private methods represent a challenge for developers writing unit tests. How should one invoke the method in the 'act' part of a unit test? The answer is simple: call them the way the program uses them, through a public method. Private methods never exist in isolation. It is always best to default to testing the unit in an environment and in a way that is as close to the program as possible. A private method will

always be accessible through a public method. Because the program will always call the private method this way, it is useful to consider the private method as a part of the same unit as the public method [27].

4.4 Scope of Testing

As previously covered, The Horizon Simulation Framework consists of a modeling and a simulation or scheduler section. These two sections are independent; the scheduler needs no information about the model, and vice versa. The models are constructed by users for their individual use case and can involve many types of systems in many domains. Several models across different domains have been constructed by previous students. Some of these models rely completely on the templates included in Horizon, while others used python to construct custom dynamic models. Because there are endless possibilities for constructing a model, it was decided that users shall bear responsibility for the correctness and compatibility of their model and Horizon shall be responsible for a reliable scheduler. This defines the scope of the tests written for Horizon. Subsystem templates, like ADCS and Power, environments, like Standard-Atmosphere and World Magnetic Model, and select methods in utilities were omitted from testing and code coverage. This omits approximately 3,000 total lines of code from the 11,000 total lines, or about 1,500 of 5,500 coverable lines and excluded 20 classes out of 142. While testing these sections was deemed not strictly necessary for Horizon's open source release, they should still be verified in the future.

4.5 Quantifying Test Suite Thoroughness

All test suites are not made equal, nor do they provide the equal utility. While some testing is better than none, the more thoroughly a software is tested, the more

useful it is to a developer for maintaining code integrity. So, to ensure that code is sufficiently tested, its thoroughness must be quantified and analyzed. There are many ways to analyze the thoroughness of a test suite. The most obvious method would be to count the methods tested and aim to test them all. This method, however, doesn't capture how thoroughly each method is tested. Most methods have multiple possible outcomes, logical paths through the method, and edge cases, all of which should be tested. While there is no way to completely and exactly define the thoroughness of the test coverage, there are many useful metrics that a code coverage tool provides which make it a superior method to counting tests or tracking methods tested.

4.5.1 Code Coverage

Code coverage tools show developers the degree to which a program is tested. It finds the areas which have not been covered by test cases. This can be quantified by statement, decision, loop, branch, condition, or finite state machine coverage [28]. A high percentage of code coverage results in a low likelihood of discovering bugs later in development. While it seems that developers would always aim for 100% coverage, this may lead to increased cost, development time and complexity, which outweigh the benefits. Instead, a goal of 80% coverage is more generally recommended [29]. Coverage goals can be a distraction developers, especially in the early stages of testing, because it might encourage covering each line rather than focusing on each use case or requirement of the software. Therefore, a goal like 80% is good to aim to avoid obsessively covering each line at the expense of development time or testing suite size. Additionally, reaching a coverage goal does not guarantee that all use cases are tested. It is highly recommended to examine uncovered lines if the coverage tool has the capability.

4.5.2 Fine Code Coverage

Fine Code Coverage, an open-source code coverage tool, was used to analyze Horizon. It displays statement coverage as a percent of lines executed during tests to total 'coverable lines', and branch coverage as branches explored to total branches for each class and assembly. 'Coverable lines' are a subset of all lines of code in a project, only including statements which can be executed. As previously mentioned, classes which are primarily for constructing models were excluded from coverage. This includes much of the HSFUniverse, which contains equations of motion and environmental models (magnetic, wind, atmosphere), and most of HSFSystem, which contains the subsystem templates. Additionally, the unit testing code itself is excluded from coverage analysis because that data does not give information about the amount of source code covered by the test data. Rather, it dilutes the coverage percentage with irrelevant data. With the ability to exclude certain classes, intuitive results display, both as a report and integrated in Visual Studio, and its responsive development team, Fine Code Coverage is a great open source tool to analyze testing thoroughness.

4.6 Continuous Integration Implementation

As discussed previously, an important feature when inviting community development with limited active moderation from core designers, as is the case with Horizon, is implementation of a continuous integration pipeline. The CI is run when users want to incorporate their changes to the main code that they forked a copy from. To implement CI alongside my testing, I set up a CI server. When deciding on the server, great importance was placed on the ease of implementation and maintenance because it will be maintained by countless others. The greatest utility is provided by a working CI system rather than a more capable but complex CI system which

is poorly maintained. Therefore, I chose a CI tool integrated with HSF source code manager, GitHub Actions. While some have noted that GitHub Actions lack some functionality of the industry standard tools, there is an obvious benefit to the tool which is integrated into the version control.

Enabling CI in GitHub requires enabling GitHub Actions, tweaking the default workflow to the desired CI workflow, linking any tests or external IO, and pushing to main. When set up correctly, an event, such as a push, pull request or external trigger, will trigger the workflow. With relatively simple commands in the workflow file, the CI server downloads necessary packages, builds the program on any of the most common operating systems, and interact with the program using provided input files. The required files include the YAML workflow file, like Horizon's in Fig. 4.5, and the test files. Once set up, the process to change the code will follow the flowchart in Fig. 4.6. In Horizon's implementation, the workflow is set up to simply arrange the necessary files, build the program, and then run the entire unit and integration test suite. If any of these steps fail, for example, a bug prevents the program from compiling, or a new feature breaks old functionality, the test results show a failure and the pull request is likely to be rejected by a moderator.

4.7 Summary of the methodology

To implement the ultimate goal of continuous integration a software must be thoroughly tested and a CI pipeline must be arranged. A series of widely accepted best practices were used as guidelines for writing these tests. Visual Studio, NUnit testing framework, and Fine Code Coverage were useful tools in facilitate writing, tracking, analyzing tests. By using these tools and methods, more time could be spent writing effective tests rather than creating the infrastructure. The unit/integration test suite

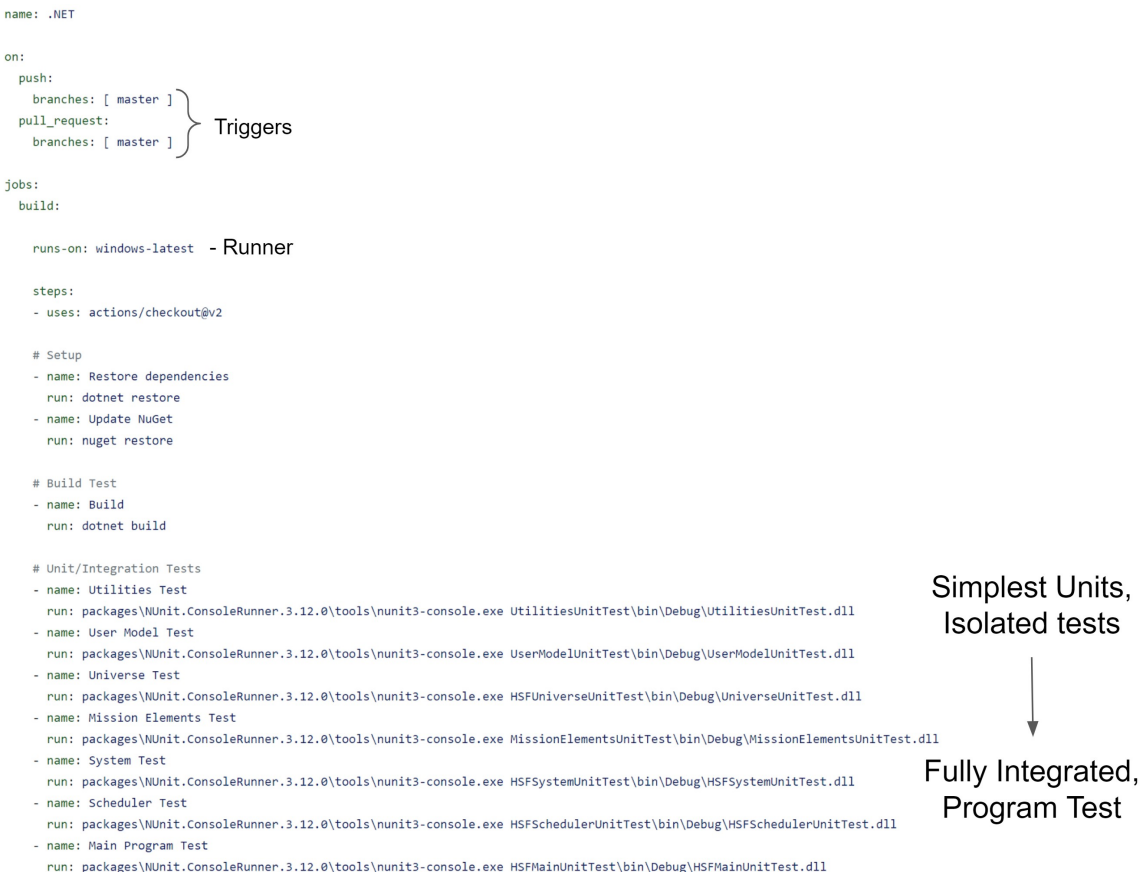

Figure 4.5: Horizon's YAML file with annotations

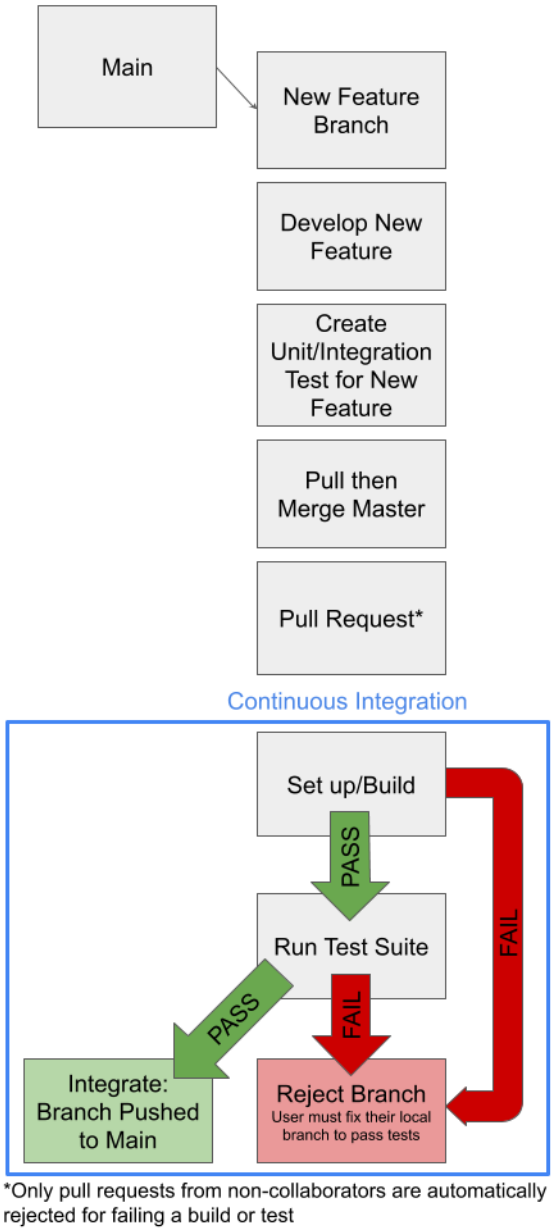

Figure 4.6: Flowchart of steps required to commit new code after CI implementation

enabled implementation of CI. Using a GitHub Actions for Horizon's CI server, a workflow was setup to automatically build the code and execute the test suite on any commit to the repository. The next chapter will discuss the results of this effort, including analysis of the test suite and CI speeds and test suite thoroughness, as well as documentation and repository improvements.

Chapter 5

RESULTS

In anticipation for its open source release, tests were written to thoroughly cover the scheduling segment of the Horizon Simulation Framework. Metrics about the tests are discussed in this chapter. The metrics analyzed are the time to run the test suite and the amount of Horizon the tests cover. The updated GitHub page and resources are also briefly discussed at the end of this chapter.

5.1 Test Speed Performance

As discussed in section 4.3.1, tests should run quickly. To achieve this speed, I used about 40 custom input files. This saved time loading unnecessarily complicated models. It also allowed the simulation duration to be customized for the needs of each test. As a result, the 126 tests were successfully run in 5.7 seconds, seen at the bottom of Fig. 4.4. This speed is specific to my local PC, which runs Visual Studio 2019 Community Edition on Windows 10 with an 8 core processor and 16 GB of RAM. On GitHub's servers, the test suite completes, on average, in 27 seconds.

5.2 Coverage Results

As discussed earlier, a coverage report gives quantitative feedback on the testing thoroughness. While no assemblies received 100% coverage, this was not the goal, and the section of the code which corresponds to the scheduling algorithm was nearly completely tested.

5.2.1 Summary Results

The test suite covers 80.9% of the coverable lines and 70.6% of branches. For context, the Aeolus test mission uses 58.1% of Horizon's source code. The uncovered portion undoubtedly contains valuable code; however, most referenced methods were addressed. I estimate that of the 19.1% remaining lines, only a tiny fraction is within the scope of this project and contain a reference elsewhere in the code. The remaining uncovered code is comprised of parts of code which are never referenced by any other part of the code and are not used by the Aeolus test mission. For example, there are some overloaded methods with less common data types, as well as extra unreferenced constructors. These methods were the lowest priority in my scope, and some remain untested.

Generated on:	8/28/2021 - 12:14:54 PM
Parser:	CoberturaParser
Assemblies:	8
Classes:	122
Files:	88
Covered lines:	3238
Uncovered lines:	761
Coverable lines:	3999
Total lines:	8271
Line coverage:	80.9% (3238 of 3999)
Covered branches:	933
Total branches:	1321
Branch coverage:	70.6% (933 of 1321)

Figure 5.1: Code Coverage Summary after executing test suite

5.2.2 Assembly Level Results

It is important that the coverage is evenly distributed through the program. If the scheduler is fully tested but no testing exists for the objects it uses in the system and mission elements, then the coverage is misleading. The covered lines as a percent of total coverable lines are shown in both figures 5.2 and 5.3. The goal of 80% coverage is displayed as the dotted line in Fig. 5.3. The three assemblies which fall just short of the goal, HSFSystem, HSFUniverse, and Utilities are associated with the modeling section, rather than the scheduling section. In contrast, the most covered assemblies, HSFScheduler, HorizonMain, and MissionElements, are the core constituents to the scheduling portion of Horizon. The number of implemented tests per assembly is shown in Fig. 5.4.

Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
+ Horizon	211	7	218	328	96.7%	39	46	84.7%
+ HSFScheduler	498	56	554	1121	89.8%	108	140	77.1%
+ MissionElements	455	71	526	1094	86.5%	129	178	72.4%
+ UserModel	80	20	100	229	80%	4	8	50%
+ HSFUniverse	302	91	393	963	76.8%	58	116	50%
+ Utilities	1390	421	1811	6644	76.7%	478	656	72.8%
+ HSFSystem	302	95	397	824	76%	117	177	66.1%

Figure 5.2: Fine Code Coverage breakdown of each assembly

55% of the remaining uncovered code exists in the Utilities assembly. This is seen clearly in Fig. 5.5, where the uncovered portion of Utilities is larger than entire assemblies. The Utilities assembly contains functionality from many previous simulation scenarios, which are unused by Aeolus. A small future project will involve deciding which unused utilities may assist future users, and which can be removed to simplify the architecture and reduce program size.

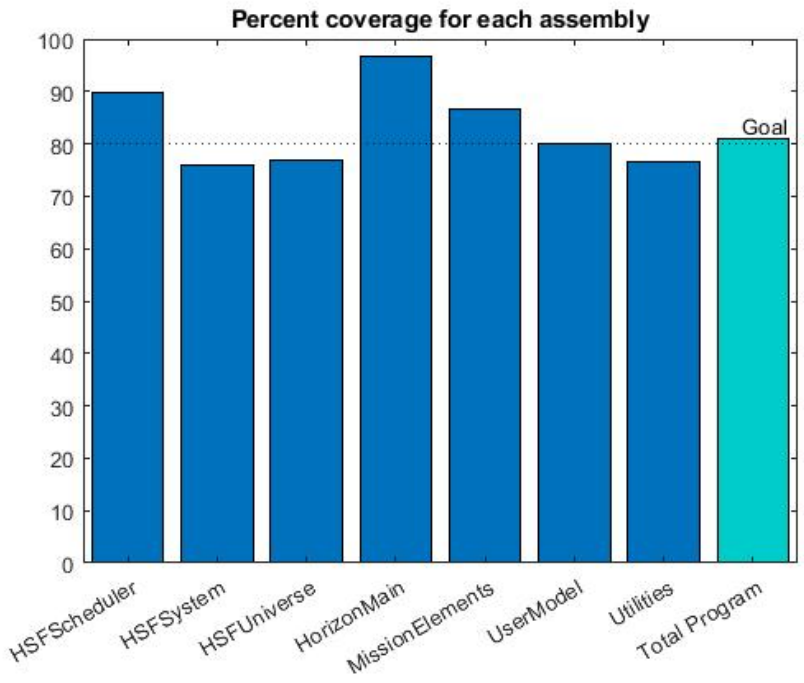

Figure 5.3: Percent coverage for individual assemblies

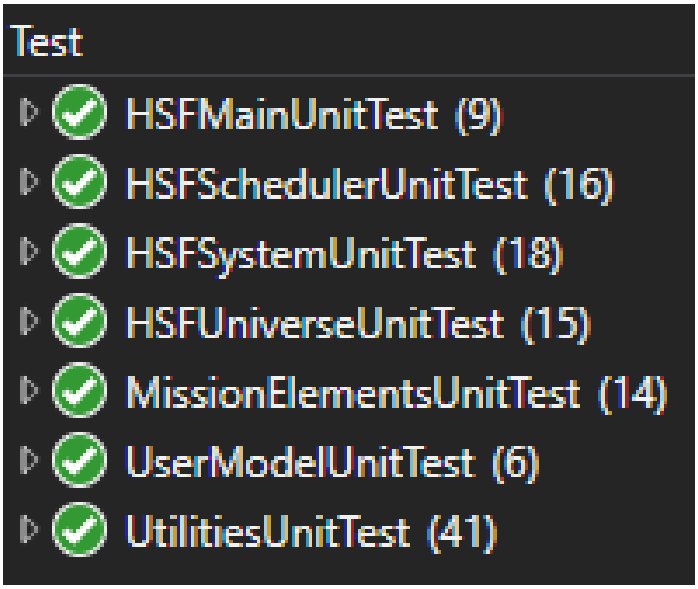

Figure 5.4: Number of tests in each assembly

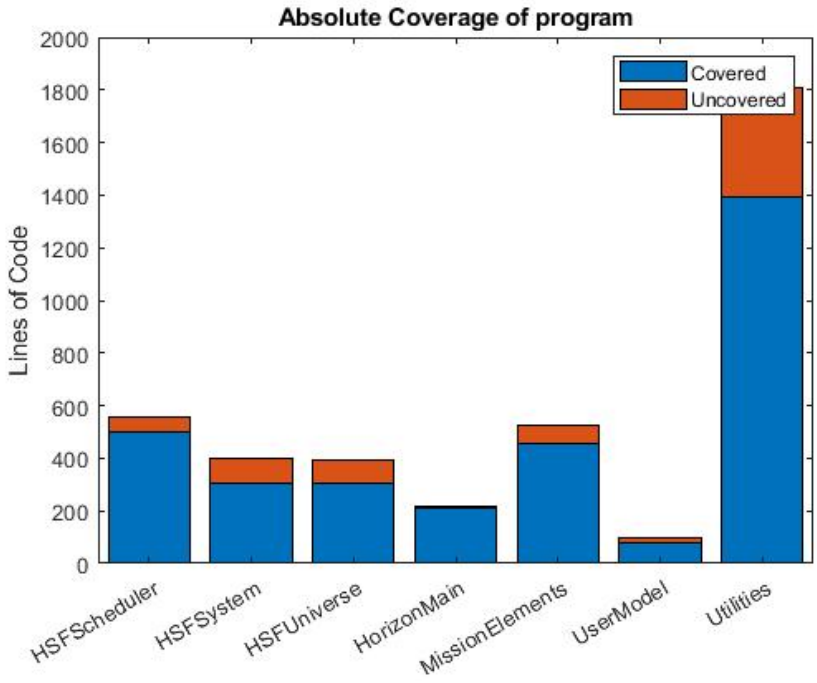

Figure 5.5: Line coverage for individual assemblies

5.3 Continuous Integration

Horizon's CI pipeline incorporates many best practices previously described by Fowler. The software remains on a single repository and now uses automated build and testing with GitHub Actions and the NUnit testing framework. For each commit, the CI workflow is automatically triggered on the CI server. This allows developers to fix commits with a broken build before it is able to be rejected. The build is relatively fast, due to the relatively small source code base and non-exhaustive test suite. For its size and development activity, the build and test execution time is sufficiently quick. Users also can find the executable and the entire history of integration and build history on the homepage/source-code manager, GitHub. This satisfies all the best practices for setting up a CI pipeline. There are other practices that relate more to behavior of developers which is beyond the control of these systems and

will rely on knowledge transfer and direct encouragement of new contributors. These are practices like committing frequently and adding tests associated with each code change. In certain cases where standards are not be followed, tests aren't written for a large new feature, or bad malicious code is discovered (e.g., removing/modifying tests which should prevent a commit) a moderator may choose to manually reject pull requests or revert to previous versions. This can be addressed by adding advanced CI features, but for a project like Horizon, the cadence of commits doesn't warrant the extra work to set this up. Overall, the elements of an effective pipeline for the Horizon Simulation Framework have been arranged.

The CI pipeline, as expected, takes slightly longer than a build and test run on my local machine. Committing a change triggers the CI workflow, which takes between 105-130 seconds, much longer than the approximately 10 second local build and test. There is only one CI path, which is triggered manually or by a push/pull request. These steps are shown in Fig. 5.6, where a failure of single test for the utilities class ended workflow and displays as a failed job.

This path includes a build on a Windows server (currently the only OS which HSF supports) and then execution of the entire test suite, which takes about as long as the local test execution. Reading the instructional file, restoring dependencies, updating NuGet packages, and post job tear-down accounts for the extra minute and a half. It is noteworthy that almost half of this added time (one third of total execution time) is spent updating dependencies and NuGet packages. Performance may be improved by caching NuGet dependencies, which nearly eliminates this step [30]. Additionally, test prioritization and optimization, as previously stated, can improve integration speed. However, this is an unnecessary complication because the speed of the current pipeline presents no current issues.

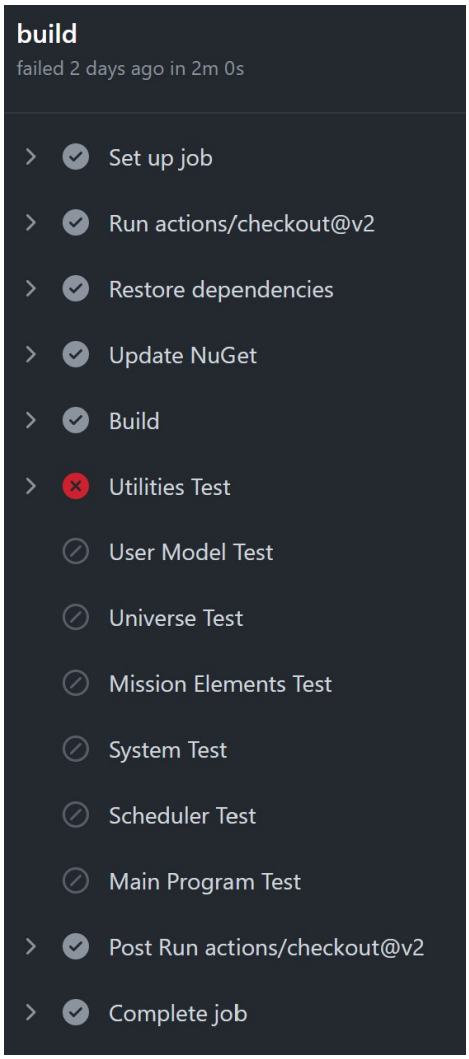

Figure 5.6: Steps taken for each commit by the CI server

5.4 Horizon's Homepage

The GitHub homepage for HSF's repository is the de facto homepage for the software. All available resources must be accessible from this page. The tools provided by GitHub, like the wiki, README, and projects pages, are sufficient to house and organize nearly all relevant information. It is also a convenient community forum to discuss issues and receive help from other users of Horizon. These resources are helpful to all users of Horizon: prospective masters students, current students, new and returning users, and developers. The appearance has also been cleaned up to make the repository more inviting and professional. This involved fixing the flat file structure and removing and categorizing out of place files.

5.4.1 Wiki

Wikis, deriving from the Hawaiian word for quick, generally organize summaries for quick reference. Wikis rely on users, typically a handful of dedicated experts, to write and organize these summaries. The Horizon wiki contains a reference page with links to graduate theses which chronicles most of Horizon's development until its open source release. It also contains a link to the user guide, written alongside the recode by Yost. Guides to help identify users' and developers' first project are also present in their own wiki tab. A critical part of the wiki are guides to standing up a user's first simulation, how to add a dependency and how to add a constraint. As Horizon's community and functionality grows, the maintenance team must continue updating the wiki and encourage community contribution.

5.4.2 README

The README.md is a file displayed on the home page of a repository. As opposed to the wealth of information hidden behind the tabs of the wiki, a README should contain short sections with only highly relevant information. A good README will tell new users what the program is, why it exists, and how to use it. Users will now find a brief introduction to Horizon, the intended use of Horizon, the installation instructions, and where to find more information. Additionally, it contains short sections such as developer and maintainer credits, licensing info, a code of conduct, and contribution guidelines.

5.4.3 Projects and Issues tabs

When a potential user or developer is ready to begin working with Horizon, they are directed to the projects or issues tabs. The projects tab contains a list of desired projects. These can be ideas for additional model libraries, feature requests, and mission concepts. This list of useful projects is designed for users without a clear idea of how they'd like to participate in Horizon's development. Similarly, issues such as bugs, poorly implemented code, or incomplete features should be reported in the issues tab. This tab is useful for tracking Horizon's development towards a more stable product. It also is a great resource for a developer looking to improve Horizon's core architecture.

5.4.4 Summary of Horizon's Resources

With these resources, visitors to Horizon's homepage should be able to find any information relating to Horizon. Users can learn more about Horizon and its capabilities

to decide whether it fits their application needs. They can also read guides to creating new models and simulations. Prospective master's students will find a comprehensive list of previous papers and ideas for future projects. Finally, developers will find a list of issues and feature requests that can inspire their own contribution to the Horizon Simulation Framework.

5.4.5 What This Work Enables

There are several students beginning their thesis with Horizon, who will all simultaneously develop new features for Horizon. With multiple students developing new functions, squashing old bugs, and reintroducing legacy features, risk of regression is high. The CI implemented in this paper will assist these students in mitigating regression, providing a quick check as to whether their push breaks some core functionality. If this is not controlled, a regressed version of Horizon might be pushed to the repository, leading to delays in all contributors' development.

Maintaining software can be a massive cost on an organization. Because the developers of Horizon are not paid employees but rather graduate students, this cost takes the form of time spent fixing old code, time that could have been spent on valuable features or research. Regression might require someone to pour through past changes to the repository and even download multiple versions to test against each other for output consistency. This person tasked with this might not have written the code causing the bug, wasting more time understanding the code prior to debugging it. By facilitating student development, CI accelerates the internal development of the program. Additionally, Horizon has finally received the testing it needed to confirm its stated functionality. This was the primary goal of this thesis.

An important secondary goal, especially for attracting an open source user/developer community, was to make the HSF's homepage a more welcoming space. By adding the necessary documents which all open source projects should have, Horizon appears more professional and worthy of a prospective developer's time. When a user looks deeper, they will no longer find a sea of nearly empty pages, but rather a wealth of organized information. With these achievements, Horizon is ready for an official open source release.

Chapter 6

CONCLUSIONS AND FUTURE WORKS

6.1 Conclusion

The goal of this project was to improve the Horizon Simulation Framework by validating the scheduling segment and releasing the software as an open source project to increase the user and developer base. To achieve this goal, I worked to verify, organize, and maintain Horizon.

To verify, I tested each relevant method using the best practices for top down integration testing and unit testing. This process led to the discovery and fixing of minor bugs and minor improvements for some functions' readability and simplicity. Unit testing also increases the perceived legitimacy and reliability of the software, imperative for any aerospace software.

To ensure the software remains verified without active moderation, I created a CI pipeline. The pipeline uses GitHub Actions to automate these tests to give a test report for each new commit to Horizon's repository. This has shown to reduce the time a commit spends in review and helps the limited moderation team effectively assess each new addition [26].

Finally, I organized the repository and added missing documentation, which made the project look incomplete and less reliable to potential users and contributors. The organization effort included adding documentation like an open source license, README.md, contribution guidelines, and a wiki. The added documentation and the pipeline will attract users by providing assurance that Horizon is a functional and

professional software. The software is now better verified, organized, has a system to maintain its integrity, all critical qualities when trying to attract new users to a new open source release.

6.2 Lessons Learned

This project gave me a deep appreciation for the work that goes into the stable software. Stable software is not achieved by a single comb through for bugs, but rather the creation of systems and workflows that maintain the integrity of the core architecture. In my future endeavors, I will use my knowledge of the importance of testing early and often and have the foresight to write tests with best practices which work well in a CI pipeline.

I learned to interact with a large code architecture by testing one function at a time. Each time I wrote a test which failed, I learned more about the flow of information and objects through the program. This is an excellent way to become intimately familiar with design of a software's architecture. Writing user or high-level functional tests, which only consider the in and out flow of data from a user's perspective, exposes the tester to only a fraction of the code architecture. Low-level unit testing provides a more complete understanding of every function and object. If I come across an untested piece of software that I must learn, I will write unit tests for the architecture, both learning from hands on development and improving the code.

Visual Studio is an excellent tool to learn C# with. Autocompletion and suggestions through Visual Studio's Intellisense helps new users asses their options for what data and methods are accessible. A user can tell if an object is inaccessible simply from its omission in the live recommendations. Another embedded tool in Visual Studio is

the compile time feedback. This feature points out hard to catch syntax and spelling mistakes as users type, rather than in an error log after a build is attempted.

Another helpful tool is GitHub Actions and workflows. This is incredibly powerful for quickly automating processes into a version control tool like GitHub. Actions can be used for more than just running tests upon each push or pull request. A workflow can be created to call a script which scrapes online sources to update a model library, or to send a change report to select contributors. There are countless ways to implement automation with the simple workflow file. Workflow files are easy to create by modifying templates, and, because it is built into GitHub, implementation is seamless.

Fine Code Coverage tool is very useful for testing more than a few units of code. FCC shows exactly which lines are uncovered by tests, so they can be addressed. It is also useful to quantify one's progress when testing. Code coverage statistics may then be accessed by other programs to produce additional reports or enforce minimum test coverage. It may even be accessed by a workflow file to prevent untested classes from being added to the repository.

6.3 Future Works

There are several features which have received countless hundreds of hours of work from previous students, only to be forgotten and fall out of compatibility with the framework. Some of these features regressed during the re-code by Yost, due to the immense scope of work covered by her project, and some come from partially completed projects or other circumstances. For adding utility, these features are low hanging fruit, meaning they add high functionality for relatively little effort. Because

of this, they are highlighted in the GitHub projects tab as great beginner projects to get familiar with contributing to Horizon.

6.3.1 GUI

The GUI is one of the most important feature additions for making the framework more accessible to those without development experience. Recall, an attractive feature of using a systems engineering modeling and simulation tool like HSF, instead of a custom-built compiled program, is that domain experts are able to define a model and run simulations without the need of a software expert. By obscuring the inner workings of the program, a GUI enables designers to focus more on model and simulation definitions. A GUI was created using Picasso by Butler in 2007, however, it was not updated to include scripting or to interact with the new C# framework.

6.3.2 More Modeling Templates

New model templates are a highly useful addition to Horizon. A primary goal for Horizon, at this stage of development, is to get the program in the hands of users. Therefore, the Horizon team welcomes any addition which lowers a barrier of entry. Making a model from a template or modifying an existing one is much quicker and easier than writing custom classes for a model. While requiring the user to first build custom functions may be inevitable for projects with a high degree of specialization, some aspects of modeling can be nearly exhausted with only a few additional models. For example, atmospheric, space, and oceanic or aquatic environmental models are sufficient to describe a majority of physical scenarios.

For most cases, defining subsystems is the most laborious aspect of modeling a system of systems with Horizon. A system has likely one domain, one set of governing

equations, and maybe a few utility classes to support these. However, a system of systems by definition has multiple subsystems, with the typical space mission containing 5-15 subsystems. Another tool to reduce the barrier of entry for new users would be a bank of presets for subsystem templates. Python scripts can be written to scrape web pages for COTS subsystem specifications. Users would then have the choice to define their own subsystem or choose from a list of existing products to construct their model.

6.3.3 Testing Modeling Segment

The correctness of the models relies on the accuracy of its parameters, the class structures and their underlying state machine functions. The accuracy of parameters is mostly unreasonable to test, outside of specific cases like catching physically impossible values. However, the class structure and functionality of subsystems and domain templates can and should be unit tested. For example, the power subsystem template can be tested for its ability to assign parameters from an input file, that the current charge changes when power is drawn, and that it returns that it can't perform a task which drains the battery more than the max depth of discharge. Many new users will opt to build a model with the given templates due to its relative ease when compared to writing a suite of new classes. Horizon should therefore test as many model template files as possible and clearly label those files which are not tested or verified.

6.3.4 Testing Guidelines and Enforcement

The CI pipeline gives users an objective measure as to whether the current state of the program works according to the original intended function. While protecting the existing core scheduling algorithm is a great start, the team expects new features

to be added to both the scheduler and the modeling section. When this happens, the scheduler no longer carries the guaranteed performance that this project aimed to provide. Developers are strongly encouraged to include unit tests with any new modules. Guidelines for where and how to do this is present in Horizon's wiki. In the future, enforcement may be implemented to reject any module without a certain percentage of test coverage. This might be performed by using a workflow which checks a coverage report that any new classes meet a minimum coverage of their tests, or that overall coverage has not fallen below a minimum value.

Furthermore, the existing tests and CI testing framework should be protected from edits from anyone who isn't a core framework developer. Currently, there is nothing stopping a user from altering a test to pass or completely removing a test which is broken by a commit. While the repository is still actively managed, meaning Dr. Mehiel must approve pull requests from outside contributors, there still exists a possibility that a small change to a unit test goes unnoticed or a unit test is incorrectly changed by a trusted contributor. Protection against this can be added with GitHub secrets, which restricts the editing of certain files (e.g., workflow file, unit tests) to those with the secret code.

Bibliography

[1] *GitHub - emehiel/Horizon: Morgan's Thesis*. URL: https://github.com/emehiel/Horizon.

[2] *SysML Open Source Project - What is SysML? Who created it?* URL: https://sysml.org/.

[3] *Final Report of the Model Based Engineering (MBE) Subcommittee*.

[4] Zane Scott and David Long. *A Primer for Model-Based Systems Engineering*. Vitech, Oct. 2011.

[5] B. Selic. "The pragmatics of model-driven development". In: *IEEE Software* 20.5 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[6] *Clarus Concept of Operations, ITS Report*. URL: https://web.archive.org/web/20090705102900/http:/www.itsdocs.fhwa.dot.gov/jpodocs/repts_te/14158.htm.

[7] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[8] "The Current State of Model Based Systems Engineering: Results from the OMG™ SysML Request for Information 2009". In: (2010).

[9] Matthew Hause. "An Overview of the OMG Systems Modeling Language". In: *Embedded Computing Design* (Aug. 2007).

[10] H Mehrpouyan et al. "Formal verification of complex systems based on SysML functional requirements". In: Proceedings of the Annual Conference of the Prognostics and Health Management Society 2014. 2014, pp. 176–187.

[11] *GitHub - Systems-Modeling/SysML-v2-Pilot-Implementation: Proof-of-concept pilot implementation of the SysML v2 textual notation and visualization.* URL: https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation.

[12] *About OSATE- OSATE 2.9.0 documentation.* URL: https://osate.org/about-osate.html#osate-modeling-capabilities.

[13] *GitHub - osate/osate2: Open Source AADL2 Tool Environment.* URL: https://github.com/osate/osate2.

[14] *Systems Tool Kit.* URL: https://www.agi.com/products/stk.

[15] David Fishering. *An Assessment of the Benefits Associated with Software By Analytical Graphics Inc.* URL: http://p.widencdn.net/gehfgb/Frost-and-Sullivan-ROI-AGI-Software.

[16] "JSBSim: An Open Source Flight Dynamics Model in C++". In: *AIAA Modeling and Simulation Technology Conference and Exhibit* (Aug. 2004).

[17] *GitHub - MASTmultiphysics/mast-multiphysics: Multidisciplinary-design Adaptation and Sensitivity Toolkit (MAST) - Sensitivity-enabled multiphysics FEA for design.* URL: https://github.com/MASTmultiphysics/mast-multiphysics.

[18] *FlightGear - Flight Simulator.* URL: https://sourceforge.net/projects/flightgear/.

[19] "SU2: An Open-Source Suite for Multiphysics Simulation and Design". In: *AIAA Journal* (Mar. 2016).

[20] *GitHub - su2code/SU2: SU2: An Open-Source Suite for Multiphysics Simulation and Design.* URL: https://github.com/su2code/SU2.

[21] Joseph Feller et al. *Perspectives on Free and Open Source Software*. The MIT Press, 2007.

[22] *Starting an Open Source Project*. URL: https://opensource.guide/starting-a-project/.

[23] Martin Fowler. *Continuous Integration*. May 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[24] Sebastian Elbaum, Gregg Rothermel, and John Penix. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". In: New York, NY, USA: Association for Computing Machinery, 2014. ISBN: 9781450330565. DOI: 10.1145/2635868.2635910.

[25] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. "Communicating continuous integration servers for increasing effectiveness of automated testing". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 374–377. DOI: 10.1145/2351676.2351751.

[26] Bogdan Vasilescu, Yue Yu, et al. "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub". In: Sept. 2015.

[27] Tom Dykstra et al. *Unit testing best practices with .NET Core and .NET Standard*. July 2018. URL: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices.

[28] Thomas Hamilton. Aug. 2021. URL: https://www.guru99.com/unit-testing-guide.html.

[29] Sten Pittet. *An introduction to code coverage*. URL: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[30] *Building and testing .NET*. URL: https://docs.github.com/en/actions/guides/building-and-testing-net.

[31] *Cal Poly Github*. URL: http://www.github.com/CalPoly.

[32] Cory O'Connor, Eric Mehiel, and Brian Butler. "Horizon 2.1: A Space System Simulation Framework". In: Aug. 2008. ISBN: 978-1-62410-000-0. DOI: 10.2514/6.2008-6546.

[33] Morgan Yost. *An Iteration on the Horizon Simulation Framework to Include .NET and Python Scripting.* June 2016. DOI: 10.15368/theses.2016.77.

[34] Cory O'Connor and Eric Mehiel. *Horizon: A System-of-Systems Simulation Framework*. May 2007. DOI: 10.2514/6.2007-2937.

[35] Brian Butler. *Dynamic Model Creation and Scripting Support in the Horizon Simulation Framework*. Feb. 2012.

[36] Brian Wagner and Steve Berdy. *Classes and Structs - C# Programming Guide*. May 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/.

[37] Gaurav Duggal and Mrs Bharti Suri. *Understanding Regression Testing Techniques*. 2008.

[38] *GitHub - JSBSim-Team/jsbsim: An open source flight dynamics control software library*. URL: https://github.com/JSBSim-Team/jsbsim.

[39] Nwillc. *Travis CI to GitHub Actions*. URL: https://nwillc.medium.com/travis-ci-to-github-actions-925d574f3f2b.

APPENDICES

Appendix A

BACKGROUND ON THE HORIZON SIMULATION FRAMEWORK

This appendix provides a more detailed explanation of Horizon's architecture design, its history, and its capabilities and feature extensions. For further reading, see the papers cited in this section or Horizon's wiki [1] found on the GitHub.

A.1 Horizon's Latest Version

The latest version of Horizon is stable and will likely be the final form of the architecture. While there are active projects to tweak certain relationships and communications between modules, the design, which is detailed in this section, will likely remain as it was after the re-architecture by Yost.

A.1.1 Horizon Introduction

The Horizon Simulation Framework (HSF) is a modeling and simulation tool which takes models, targets, and constraints and outputs schedules for the system to capture targets within constraints. The latest version, HSF 3.0, is written in C# and IronPython, an implementation of Python which leverages the capabilities of .Net Framework. The primary application of Horizon has been for aerospace system modeling and simulation, as discussed in section A.2.2. Horizon requires three inputs, a target deck, models, and simulation parameters. Models are comprised of subsys-

tem templates written in C# or python and an XML input file which gives Horizon the subsystems and their parameters required to build a system model. From these inputs, Horizon returns schedules, or lists of successfully completed tasks with start and end times. Because Horizon is split into two independent components, modeling tools and a time driven scheduling algorithm, the algorithm can simulate any model in any domain, without increases in complexity. It is useful to discuss the functions of each component prior to describing how they interact

A.1.2 Modeling Segment

Prior to simulating, Horizon must first construct the model. The model input file contains the information necessary for Horizon to construct the model. HSF provides the basic elements for modeling like environments, governing equations, assets, subsystems, dependencies, initial conditions, and constraints. Some templates exist for specific implementations of environments, equations, and subsystems, but if a specific application requires any of these without a template, users must create their own. Users must specify parameters for subsystems and environments with the xml file. In the xml file, users can also instruct the program to use a python scripted template, instead of one of the provided C# template subsystems or environments. Within each subsystem's node, its characteristics, initial values, constraints, and dependencies are specified. The dependencies for each subsystem are collected while the model is stood up. These dependencies dictate which subsystems depend on another subsystem to update its own state. For example, Horizon's default communications (COMMS) subsystem is dependant on the data rate of the solid state data recorder (SSDR), and power is dependent on the state of both subsystems. This means that the state of SSDR must be evaluated first, then it passes the value through the dependency link to COMMS to determine its value, then finally both are passed to Power to set the

state. This process must always begin with a subsystem with no dependencies. This prevents circular dependencies, which are not allowed by Horizon. Dependencies are the only way subsystems may pass data to each other, and they are imperative to the correct evaluation of each subsystems state.

A core principal of the modeling segment is the state machine inside each subsystem. The state machine records the state of its subsystem as tasks are performed. Horizon ensures that the state of each subsystem is evaluated in the order specified by the dependency flow. At the heart of each subsystem is the canPerform() and canExtend() methods, which evaluate whether that subsystem can perform or extend the task. Horizon compares a subsystem's state data to its constraints, which are set by users in the model input file. If the new state breaks a constraint, the model segment returns a false to the scheduling segment, and this task-schedule pair is not added to the list of possible schedules. If no constraints are broken, all of the subsystem states are saved for that schedule, and the model returns true.

A.1.3 Scheduling Segment

The scheduling algorithm uses the model to create schedules. The scheduling algorithm passes a task to a model which the model executes, checking its canPerform if any constraints are broken by the task. If all subsystems return true when their canPerform() function is called, then the scheduler creates a copy of the current schedule and adds the performed task to create a unique schedule. Schedules are assigned a value which attempts to score a schedules usefulness in achieving mission goals. The value of a schedule is dictated by the number of targets tasked and the value of each target. The weight assigned to each target has substantial impact on the simulation results. For each time step, all tasks are given to the modeler to evaluate each subsystem's `canPerform()` or `canExtend()`. This quickly leads to a massive number of

computations per step, if unchecked. However, due to the branch and bound scheme, discussed in section **??**, a user can easily prevent this by specifying a bound. When the number of schedules exceeds the maxSchedule bound, the lowest scoring schedules are eliminated from memory. When the program finishes evaluating the given simulation time frame, all remaining schedules are written to a text output file, and commands given to subsystems to perform the highest scoring schedule are printed into separate csv files.

A.1.4 Aeolus

The mission which was constructed alongside and used to test the final compiled program is called Aeolus. It emulates the type of low fidelity modeling that might be used during the initial stages of the mission design of a space system. The objectives of the mission are to image targets around the earth and downlink data to ground stations. Aeolus achieves these objectives with 2 identical satellites containing the following subsystems: power, solid state data recorder, earth observing sensor, attitude determination and control system, and communication. Due to the relatively few inputs required to define the system, Aeolus demonstrates the ease with which a model can be stood up and quickly adjusted. A quick glance at the output of Aeolus shows that the configuration can sufficiently perform the desired objective.

A.1.5 Why is Horizon Useful?

HSF can perform three high-level Mission Design/System Engineering functions for users: Bottleneck and Leverage point analysis, functional and utility requirements verification and validation, and generating sufficient schedules to carry out a mis-

sion. This generally corresponds with three stages of system development: design, verification, validation and testing (VV&T), and implementation.

The creators of the framework were primarily concerned with the framework being both reusable and providing a high amount of utility to the user [32]. This was achieved with three concepts, modularity, flexibility, and utility, which are used at every level of the architecture design.

Modularity decreases the complexity of designing a system because discrete parts of the architecture can be more easily understood and modified. Modularity also drove the decision to separate the scheduling algorithm from the system models, allowing either to be changed without requiring adjustments to the other [32]. This is also seen in the rigid separation between subsystem models, allowing 'plug and play' of existing subsystem models. Lastly, the modularity allows the flexibility for users to simulate any domain because of the framework's ability to interchange different environmental models and governing equations modules. This flexibility is another key feature of Horizon.

Flexibility is seen in the range of model fidelity and the possible applications of the framework. The fidelity of a simulation is solely dependent on the complexity of the model. Thus the system can be modeled roughly during early design stages, and modeled at a higher fidelity during VV&T [32], a key component of MDD and MBSE. Secondly, the independence of models and scheduling algorithms allows a vast array of applications outside the original intent of space systems.

Finally providing utility is the ultimate goal of the HSF. "Libraries of integration methods, coordinate frame transformations, matrix and quaternion classes, and profile storage containers were created to separate the user from the overhead of using the... programming language [32]." Additionally, as the usage of HSF grows, so does

its libraries and thus grows its utility. HSF enhances the standard project development with MBSE best practices, starting with requirements. Once requirements are made exactifiable and objective, they can be translated into models which should completely describe the full state of the system (dynamic and otherwise). This satisfies a key pillar of MBSE which is to model requirements for traceability and consistency. Then a preliminary design can be analyzed for failures, bottlenecks, leverage points. These undesired emergent behaviors are removed with each iteration prior to converging upon a baseline design. Analyzing these failures can be even more valuable to an engineer than the successful results [33].

As the design process continues 'down the V', Fig. 1.1, to the component level, trade analysis can be performed using the existing models. HSF helps identify leverage points during this stage, which describe parameters which, when changed slightly, produce wildly different outputs. In practice, an analyst can try slightly different components or configurations which affect minimal cost on the system to find potentially large benefits to the system capabilities.

Then as the system design is locked in, validation and verification of the original requirements is performed (up the V, Fig. 1.1) updating the original models with the exact specifications of each component. Each time the design is updated, likely by many separate teams, the system can be re-evaluated to ensure that the functional requirements and the utility requirements are always both satisfied.

Finally, once baselines are locked in, and bottlenecks, leverage points, and emergent behavior have been identified and exploited, analysis for a final mission schedule can be performed with the built-in scheduler. Users can specify the time step and initial conditions as well as the bound for number of schedules kept. The team may program their system to execute a schedule from the list. However, for a dynamic mission,

a team might use Horizon to constantly find optimal schedules while an asset is in orbit, as parameters and program needs change.

A.2 Brief History of Horizon Simulation Framework

A.2.1 Beginnings with O'Connor

The current framework has kept all the core functionality (subsystem modeling, dependency, task scheduling) from the initial version. Horizon V1 was created in 2006 by students and faculty of the Cal Poly Space Technologies and Applied Research laboratory (CPSTAR) to fill the gap between generalized modeling tools and more specific visualization tools like STK. Cory O'Connor developed Horizon Simulation Framework v1.2 alongside the first test mission, Aeolus. The HSF was updated to v2.1, also by O'Connor [34], to include multi-asset modeling, essentially enabling another layer of complexity to the missions run by Horizon. HSF v2.1 also introduced a recursive algorithm for dependencies which prevented users entering circular dependencies, and ensures that dependencies are always executed in the proper order to ensure state data are available to all subsystems [34].

A.2.2 Minor Improvements and Implementations

After multi asset support was added in 2008, several implementations and improvements validated the framework design and the general concept as useful and robust. Systems which were modeled and simulated during this period include UAV thermal soaring (Li 2010), integrating an existing SysML CubeSat model with the framework (Luther 2016), a sounding rocket stabilization system (MacLean 2017), an amateur astronomy network using a constellation of CubeSats (Johnson 2018), and a UAV

swarm with cooperative control using digital pheromones (Frye 2018). Various feature upgrades include a hybrid genetic algorithm as an alternative to the branch and bound scheduler algorithm (Seibel 2009) increased orbital propagators (Farahmand 2009), a GUI (Kirkpatrick 2010), dynamic model creation and scripting support (Butler 2012), system failure analysis (Lunsford 2016), and a SysML output interface and system-level requirement analyzer (Patel 2018). These constituted minor progress to the framework compared to more comprehensive projects tackled by Butler and Yost.

A.2.3 Scripting Addition

The first of the major upgrades by Brian Butler was the addition of Lua scripting in version 2.1. This was seen as significant improvement for usability because of the simplicity of using Lua scripting to construct models compared to C++. Scripting is believed to decrease time required to stand up a model with little impact on the performance of the simulation [35]. In support of this, dynamic model creation was also added, preventing the need to recompile the framework for minor changes in the models [35]. Finally, multi-threading was added to take advantage of modern multi-core processing. These modifications ultimately change how users interact with HSF by moving the user further away from the detailed compiled code.

A.2.4 Yost's Re-code/Re-architecture

One of the most transformative projects for the framework was the work from Morgan Yost who translated the framework from C++ with Lua scripting to C# with Python scripting. Benefits of this change include general improvements from a comb-through of the old code which led to an improved, uniform coding style, a simplification and organization of the architecture framework so that future versions will not require

major restructuring, and adoption of Python, a scripting language with more users and resources than Lua [33]. Additionally HSF is now able to leverage additional functionality provided by the .NET framework, including inheriting C# objects into scripts, cross language integration, and exception handling [**wagner**]. The only depreciated feature in the update to 3.0 was multi-threading capability. This overhaul represents most of the current version of HSF and was instrumental in its evolution. Since then, a minor update by Alex Johnson made it clearer how state information is stored and shared.

A.2.5 State of Horizon

HSF has been in closed development by Cal Poly masters students since its inception in 2006. It's collection of utilities, modeling templates, algorithms, and external plugins have grown slowly over the years. Some years, multiple students develop several new features for Horizon. Sometimes, years pass with no student development on Horizon. This has led to unsteady development and a lack of a consistent vision for Horizon's direction. Another undesired result of this, features are sometimes forgotten and never used again after the thesis. This long period wait between thesis students means that Eric Mehiel, the Horizon advisor, must be the sole keeper of practical knowledge and awareness of the current state of Horizon. Many more minor features, fixes and nice-to-have's have been low priority for development because most graduate students want work on a significant feature addition. This has led to many features being underdeveloped or regressed (utilities, modeling library, subsystem modeling/cots, documentation).