Volume

2

HORIZON

A Systems-of-Systems Simulation Framework

Users Guide

Horizon: A systems-of-systems simulation Framework

Users Guide

© Cal Poly, San Luis Obispo

1 Grand Avenue • Aerospace Engineering Department

San Luis Obispo CA, 93407

Phone 805.756.7537 • Fax 805.756.2376

Table of Contents

[Release Notes iii](#_Toc461303618)

[Using Horizon iii](#_Toc461303619)

[System Requirements iii](#_Toc461303620)

[What is Horizon iii](#_Toc461303621)

[Introduction to the Horizon Simulation Framework 4](#_Toc461303622)

[Welcome 4](#_Toc461303623)

[Requirements and Modeling and Simulation 4](#_Toc461303624)

[Systems Engineering 4](#_Toc461303625)

[Feedback and Flow 5](#_Toc461303626)

[A Modular Approach to System Modeling and Simulation 5](#_Toc461303627)

[The Horizon Library of Utilities 5](#_Toc461303628)

[Getting Started 5](#_Toc461303629)

[An Example 5](#_Toc461303630)

[The Basics 5](#_Toc461303631)

[Building a System Model 7](#_Toc461303632)

[The Building Blocks 7](#_Toc461303633)

[What is State? 7](#_Toc461303634)

[Fundamentals: The StateVarKey and HSFProfile 7](#_Toc461303635)

[Targets, Tasks and Events 8](#_Toc461303636)

[Dependencies Between Subsystems 8](#_Toc461303637)

[Dependency Modeling – The Middle Man 9](#_Toc461303638)

[Built-in Dependency Function Definitions 9](#_Toc461303639)

[Dependency Collectors 10](#_Toc461303640)

[Defining a Dependency in the Framework 10](#_Toc461303641)

[Expected Behavior 12](#_Toc461303642)

[Subsystem Modeling 12](#_Toc461303643)

[The Horizon System Modeling Philosophy 12](#_Toc461303644)

[Adding a Subsystem to the Simulation 12](#_Toc461303645)

[Subsystem Attributes 12](#_Toc461303646)

[Required Fields 13](#_Toc461303647)

[Attitude, determination and Control 13](#_Toc461303648)

[communication 13](#_Toc461303649)

[Earth observing sensor 13](#_Toc461303650)

[power 14](#_Toc461303651)

[solid state data recorder 14](#_Toc461303652)

[Subsystem Child Nodes 14](#_Toc461303653)

[Initial conditions 14](#_Toc461303654)

[Dependency 15](#_Toc461303655)

[Scripted Subsystem Definition 15](#_Toc461303656)

[Inheriting from a built in subsystem 15](#_Toc461303657)

[Inheriting From an Abstract Subsystem 19](#_Toc461303658)

[Subsystem Constraints 20](#_Toc461303659)

[Creating a Custom Constraint 21](#_Toc461303660)

[Putting it All Together 21](#_Toc461303661)

[Running a Simulation 22](#_Toc461303662)

[Input 22](#_Toc461303663)

[The Model Input File 22](#_Toc461303664)

[Asset tag 22](#_Toc461303665)

[Dynamic State tag 22](#_Toc461303666)

[Subsystem tag 23](#_Toc461303667)

[Contraint tag 23](#_Toc461303668)

[Equations of motion tag 23](#_Toc461303669)

[Initial Condition tag 23](#_Toc461303670)

[Dependency tag 23](#_Toc461303671)

[Statevarkey Tag 24](#_Toc461303672)

[The Simulation Input File 24](#_Toc461303673)

[Scenario Tag 24](#_Toc461303674)

[Simulation Tag 24](#_Toc461303675)

[The Target Input File 25](#_Toc461303676)

[While Horizon is Running 26](#_Toc461303677)

[The Scheduling Algorithm 26](#_Toc461303678)

[Constraint Checking 28](#_Toc461303679)

[The Cascade Constraint Checking Sequence 28](#_Toc461303680)

[Dependencies 30](#_Toc461303681)

[Output 30](#_Toc461303682)

[Comma Separated Text Files 30](#_Toc461303683)

[Glossary 31](#_Toc461303684)

[Index 32](#_Toc461303685)

Introduction

## Release Notes

This documentation is applicable to release 3.0 beta of the Horizon Simulation Framework. The simulation framework is now implemented in C# with basic features.

To contact the Horizon Development Team with any questions, please contact:

Dr. Eric A. Mehiel

California Polytechnic State University, San Luis Obispo

Aerospace Engineering Department

1 Grand Ave.

San Luis Obispo CA, 93407

(805) 756-7537

[emehiel@calpoly.edu](mailto:emehiel@calpoly.edu)

## Using Horizon

To obtain the Horizon Simulation Framework, visit github.com/emehiel/Horizon.

### System Requirements

The Horizon Simulation Framework has been designed to work with the Windows operating system. During development, the Horizon Team at California Polytechnic State University, San Luis Obispo, (Cal Poly, SLO) used Microsoft Visual Studio Community Edition.

The hardware requirements for your computer system are really dependent on the size of the problem you are solving and whether or not you have a dedicated machine for running simulations.

## What is Horizon

The Horizon Simulation Framework is a set of Modeling Tools combined with a time driven simulation algorithm. The result of running the Horizon Framework for a given set of models is the state data associated with each system model at each time step of the simulation.

The Horizon Simulation Framework is flexible due in part to the simplicity of the main algorithm used to drive the simulation. The main algorithm’s complexity does not vary with the system being simulated. The Horizon Framework implements a modeling policy that ensures several key modeling concepts:

1. The Main Algorithm does not need information from the system model. Therefore, the Horizon Simulation Framework can simulate any system.
2. System models cannot have dependencies unless the dependencies are explicitly defined via The Horizon dependency.
3. All constraints imposed on the valid states of the system must be non-circular in time. In other words, the feasibility of a subsystem state cannot depend on future possibilities.

When these guiding principles are adhered to, the Horizon Simulation Framework provides a very flexible, simple method for simulation.

Chapter

1

Introduction to the Horizon Simulation Framework

A Systems-of-Systems Modeling and Analysis Tool

# Welcome

## Requirements and Modeling and Simulation

The satisfaction of requirements is fundamental to the engineering design process. Once an initial design is conceived, analysis is required to verify that the system specifications satisfy the mission requirements. A common, cost effective method for performing such analysis is accomplished through modeling and simulation. By simulating the mission with a model of the proposed design at a high level early in the design phase, critical design bottlenecks and short comings can be identified and corrected before the project progresses. By creating smaller scale or computer based models of a potentially costly system, the technical life cycle can be executed for a representative system at reduced overall cost. In the specific case of space mission design, modeling and simulation is especially important because of the high cost of space missions, the relatively long timeline of the project and the inaccessibility of the operational environment.

### Systems Engineering

In order to design the complex systems required for aerospace missions, systems engineering is required to decompose the large system to smaller subsystems. In this way, the engineer must design subsystems and interfaces that ultimately satisfy the mission requirements. Because simulations are effective ways to prove design feasibility, a simulation that allows for mission analysis at the subsystem level is especially useful.

Systems engineering was taken into consideration when designing the Horizon Simulation Framework, and as a result, the framework is designed to be modular. At a high level, the framework is split into two major components, the modeling component and simulation component. The simulation component, or scheduler, is completely developed and is a “black box” to the user with only mission targets exposed for modification. The interface between these two components is well defined so that the user may easily create a model to be run in the simulation. The model itself is a system that is comprised of modular subsystems. The user may modify the provided system to fit design criteria, or build a system completely from scratch.

### Feedback and Flow

A primary benefit of using computer models to simulate the behavior of the modeled system is the rapid pace that models can be generated and tested. In this way, the project can move through multiple iterations at a rate that is not detrimental to the schedule. When Horizon identifies bottlenecks and broken constraints or requirements, the improvements made to correct for these shortcomings can immediately be tested to verify that the system still functions correctly and is in-fact improved.

### A Modular Approach to System Modeling and Simulation

In order to ensure that Horizon meets the needs of all systems, the framework was designed with plug and play functionality in mind. This was accomplished by standardizing interfaces so that one element of the framework could be modified or exchanged without the need to modify all the dependent elements.

### The Horizon Library of Utilities

The Horizon Library of Utilities is provided to the user in order to allow for the use of common functions found in typical modeling software such as MATLABTM. The methods included in the Horizon Library of Utilities can be found in \*\*\*\*\*\*.

# Getting Started

## An Example

The Horizon Simulation Framework was created alongside with it’s test case—Aeolus. While the built in subsystem models and accompanying targets are specific to Aelous, small modifications can be made to model similar missions, and with an understanding of HSF, any model and mission can be synthesized and simulated.

The Aeolus mission is a simple one: image as many targets as possible and downlink the data when needed in one orbit of 6307 seconds. Aeolus is a two satellite constellation with both satellites starting on opposite ends of the globe from each other and with opposite velocity vectors. The subsystems that make up each satellite includes the Attitude, Determination and Control (ADCS), Communication (COMM), Earth Observing Sensor (EO Sensor), Solid State Data Recorder (SSDR) and Power. These subsystems were modeled as described in Chapter 3. In order to create a system from subsystems, HSF provides the basic infrastructure to define constraints on the performance of the system and inter-dependencies between subsystem, as well as maintain state data. A constraint is a limitation placed on a subsystem that dictates the functional bounds of the model. For example, the power subsystem may have a constraint that the battery depth of discharge is not allowed to surpass 70%. A dependency is an interface used to communicate data between subsystems, so that a subsystem with a dependency may calculate its state from its dependent subsystem. An example of this is that the power subsystem must know how much power the ADCS required to complete the current task. Lastly, the state is maintained at a system level with each subsystem propagating its own state. The details of these fundamental elements will be discussed in the chapters that follow.

## The Basics

Before the low level details of the Horizon Simulation Framework are discussed, a brief high level synopsis will help to keep the elements of the framework in perspective. The main task of HSF is to create the scenario for the user’s model to run in. For that reason, the framework is split into two major segments: the scheduling segment and the modeling segment. The scheduling segment relies on the user’s input of targets to attempt to make a temporal schedule of tasks for the model to execute. The scheduler is a black box to the user. The user, however, is responsible for the model. HSF provides the basic elements to create a system model such as template subsystems, dependencies and constraints, but the user must define the functionality of the model. The model is essentially a state machine that attempts to progress through each task generated by the scheduler. The state machine within each subsystem model that makes up the system model is the CanPerform method. This method takes in a task from the scheduler and propagates the state forward through the task in order to determine if the subsystem can perform the task. Whether or not the subsystem can execute the task is reported back to the scheduler in order to determine if the schedule is obtainable by the system. More details on all elements of the framework will be discussed in the chapters that follow.

Chapter

2

Building a System Model

Subsystem, Constraints and Dependencies

# The Building Blocks

## What is State?

The state of any system is the collection of variables required to fully specify the condition of that system. The change in the state variable also represents how the system changes over time. All dynamic systems have state variables and as the state vector changes over time, we can define a transition operator which governs that change. Therefore, we are typically interested in modeling the transition operator. The Horizon Simulation Framework is essentially a tool that moves the state forward in time and the job of modeling the transition operator is incumbent on the user. As such, the user must specify all state variables as part of the subsystem models and all state variables must be updated by the user designed Subsystem models.

## Fundamentals: The StateVarKey and HSFProfile

Before the development of a system can be discussed, the fundamental elements of the Horizon Simulation Framework must first be defined. From a high level, a system is composed of subsystems that are interconnected with dependencies. The state of the entire system is held in one key value pair storage system. It is useful to think of this state like a bulletin board, with subsystems posting their changes as their individual state evolves. Each subsystem manages its own state variable and is the only subsystem that has access to read or write to it. This is managed through the use of a StateVarKey. The StateVarKey is the key used to access the state of the specific subsystem from within the key value pair storage system. The privacy is obtained by only allowing subsystems to know their own StateVarKey, and encouraging the developer to not share the key under any circumstances. The value associated with the key is the actual state information of the subsystem and is stored in an HSFProfile. The StateVarKey and the HSFProfile are both templated types, and their types must match. That is, a StateVarKey<double> will be associated with a HSFProfile<double>, meaning that the subsystem is storing double values in the profile associated with that that key. The HSFProfile is another key value pair storage system to store the value of the state data over time. The key in this case is the simulation time, and the value is the state of the subsystem at the time. The HSFProfile follows a zero order hold pattern so the new state is only stored if the value changes, otherwise, the state at all subsequent times is assumed to be the last recorded state. Figure 4 is an illustration of the HSFProfile within the larger system state.

1.5

500

0.85

270

0.5

300

Time

Watts

Power Subsystem

Other Subsystems

Watts Used

State Data Bulletin Board

Event Start

Event End

0.5s

300

270

0.85s

1.5s

Intermediate Time

Watts

Figure . Representation of how the State Data is represented as a Bulletin Board in the Horizon Simulation Framework.

## Targets, Tasks and Events

The states are propagated in time chunks defined by the length of Events. Each Event can contain a task that is a command for an action towards a Target, and holds the current state of the system. All subsystems receive the same Event and it is up to the subsystem’s can perform to determine if the Task within the event applies to it, and if it can perform the task. The length of time of the events, and targets are defined in XML files as described in Chapter 3.

# Dependencies Between Subsystems

Before specifying the subsystems within the system, the dependencies between the subsystems should be fully specified by the designer. From a modeling perspective, the dependency is an interface that keeps components of one subsystem from leaking into the definition of another subsystem. Therefore, when two or more subsystems need to share information, or if one subsystem needs another subsystem model to determine an input to an algorithm, a dependency function must be used. The dependency tree is a diagram that shows how subsystems rely on one another and should be drawn for each new system in order to understand the order of execution of the subsystems. Figure 5 is an example of the dependency tree that was used to define the relationships between the built-in subsystems in the Horizon Simulation Framework. The arrows point in the direction of the dependent subsystem. This means the ADC subsystem is a dependent subsystem of the EO Sensor because the EO Sensor has a dependency on the ADC Subsystem. Similarly, the ADC, EO Sensor, SSDR and COMM subsystems are all dependent subsystems of the Power subsystem.

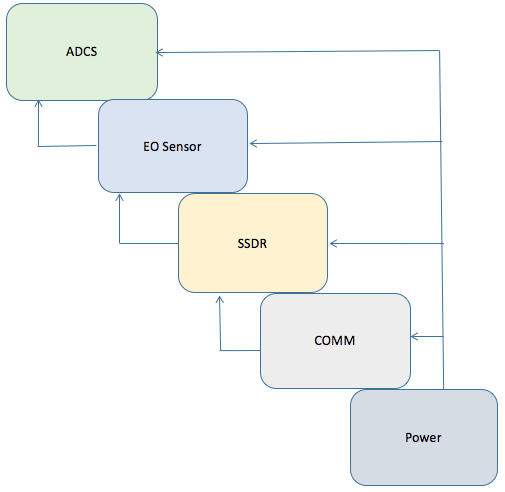


Figure . Built in Subsystems Dependency Graph

Once the dependencies are specified, the designer can begin to think about how the dependencies convey information through the dependency function. The interface between subsystems is strictly defined and enforced such that information can only be relayed between subsystems through a dependency function. As of v3.0, dependencies are extended further than previous iterations such that when subsystem state data is needed for a constraint checking, schedule evaluation or for any other HSF component, it should be relayed through a dependency function as well. This consistently enforces the rule that only subsystems should have access to their own state data.

## Dependency Modeling – The Middle Man

The dependency functions are provided as a means of exchanging information between subsystems. However, assume the power subsystem needs to know how many Watts the SSDR buffer consumed while executing a task. Since the Power subsystem is only a model of the batteries and solar panels on the spacecraft, and the SSDR model is strictly a model of memory usage, how is the power consumption determined? Enter the dependency function. The dependency function contains the algorithm which converts the state information of one subsystem into information that some other subsystem can understand and use to determine its own state. Because the state of each subsystem is private to that subsystem, the dependency function essentially calls upon the dependent subsystem to access its own state and complete the necessary conversions before relaying the requested data to the caller. All dependency functions must return an HSFProfile and take in an Event (generated by the scheduler) as a parameter. The dependency functions that are used by the built-in subsystems are described below.

### Built-in Dependency Function Definitions

Attitude Determination and Control

The ADC subsystem is a dependent subsystem of the Power subsystem so it contains the implementation of the dependency function, called POWERSUB\_PowerProfile\_ADCSSUB. This dependency function creates a HSFProfile<double> that uses the event start time and task start and end times to determine the ADCS power consumption. The power consumption is modeled such that after the event starts but before the task has begun the ADC subsystem consumes 40W, then for the duration of the task it consumes 60W, then drops back down to 40W.

Communication

The communication subsystem is also a dependent subsystem of the Power subsystem so it contains the implementation of the dependency function, called POWERSUB\_PowerProfile\_COMMSUB. The communication power consumption is simply modeled to be proportional to the data rate, so the COMM subsystem accesses its state during the event and returns the data rate multiplied by 20 in an HSFProfile<double>.

EO Sensor

The EO Sensor is a dependent subsystem of the Power subsystem as well as the SSDR subsystem so it contains the implementation of the dependency functions, called POWERSUB\_PowerProfile\_EOSENSORSUB and SSDRSUB\_NewDataProfile\_EOSENSORSUB. The power dependency function simply creates a HSFProfile<doble> where the power consumption is 10W when the EO Sensor is off and 60W when it is on. The SSDR dependency function is modeled proportionally to the number of pixels captured. The EO Sensor accesses its state for the event and returns the number of pixels captured divided by 500 in an HSFProfile<double> as the number of bytes consumed.

Power

The power subsystem is not a dependent subsystem of any of the other subsystems.

Solid State Data Recorder

The SSDR is a dependent subsystem of the Power subsystem as well as the Communication subsystem so it contains the implementation of the dependency functions, called POWERSUB\_PowerProfile\_SSDRSUB and COMMSUB\_DataRateProfile\_SSDRSUB. The Power dependency function simply returns a HSFProfile<double> with a power consumption of 15W for the duration of the event. The COMM dependency function models the data rate by multiplying the data buffer fill ratio by 5000. The data rate is returned in a HSFPofile<double>.

## Dependency Collectors

The dependency collector uses all the dependency function call keys stored in the subsystem to call all the dependency functions, then simply sums their results. Currently, all the dependency functions return HSFProfile<double>, so the dependency collector returns a HSFProfile<double>, this could however be changed by overriding the dependency collector with a different one and using that instead. The dependency collector is called from within the subsystem’s CanPerform method and used as needed to update the subsystem’s state.

### Defining a Dependency in the Framework

Dependency functions are maintained in the framework in a globally accessible key value pair storage system, called a dictionary. This dictionary stores the dependency functions and correlates them to their call key. The subsystems each maintain a list of call keys that allow them to find the dependency functions they need from the global dictionary. Of course, the global dictionary somehow needs to be populated with all the call keys and dependency functions of the system. The HSF standard for populating the dictionary is that the dependent subsystem is required to post its dependency functions and corresponding call keys to the dictionary. As an example, a dependent subsystem of the power subsystem is the SSDR subsystem in that the SSDR subsystem must relay information to the Power subsystem before the Power subsystem can determine its own state. As per HSF standards, the SSDR subsystem should have the implementation of the dependency function defined within its class and is responsible for posting this dependency function to the global dictionary. The power subsystem then just needs to know the call key the SSDR subsystem used to store the dependency function in the dictionary and it has a way to receive information from the SSDR subsystem. The Power subsystem is notified of the dependency call key in the XML file when the dependency is declared. The previous explanation describes the process used by the C# code, but adding dependency functions from scripted subsystems is identical and the steps are outlined below.

1. **Inherit the Subsystem Responsible for Defining the Dependent Function.** If the subsystem is a built in subsystem, locate the template python file for inheriting the subsystem. If you are creating a new subsystem, follow the steps for creating a new subsystem model described in the following section.
2. **Add a Dependency Function Implementation or Modify the Existing One.** If all you want to do is change the way the SSDR communicates data to the Power subsystem (i.e. the SSDR subsystem you’re modeling uses more Watts per bytes stored than the built-in SSDR subsystem) then all you must do is change the implementation of the SSDR to Power subsystem dependency function in the scripted SSDR template file. If your system, however, has a dependency between subsystems that is not in the built in dependencies or you’re making your entire system from scratch, then the dependency function must be implemented in the python file in a new function. The GetDependencyDictionary function is responsible for ensuring all dependency functions get posted to the global dictionary for subsystems with dependencies to use. Every dependency function must be added to the global dictionary by adding the following 2 lines of code to the GetDependencydictionary function for every new dependency function. The <Type> should be replaced with the type of the HSFProfile that is returned from the dependency function, the <Function Name> is the name of the dependency function as it appears in the file and the <Call Key> is the call key that is provided to the function with the dependency in the XML file.

depFunc1 = Func[Event, Utilities.HSFProfile[<Type>]](self.<Function Name>)

dep.Add("<Call Key>", depFunc1)

1. **Establishing the Dependency.** Once the dependency has been created in code, the subsystems must be notified it exists. This is accomplished in the XML file by adding a dependency child node to the subsystem with the dependency. The dependency can simply be an *order dependency* in which the dependency just forces that one subsystem be executed first. An order dependency just notifies a subsystem of its dependent subsystems as seen below.

<DEPENDENCY subsystemName="<Sub Name>"></DEPENDENCY>

Another type of dependency is a *functional dependency*. A functional dependency not only specifies order, but it specifies that there is information that needs to be relayed from the dependent subsystem through a dependency function. The child node to specify a functional dependency is shown below.

<DEPENDENCY subsystemName="<Sub Name>" fcnName="<Call Key>"></DEPENDENCY>

A subsystem can have any amount of dependency child nodes, so long as there are no circular dependencies. An error will occur if a subsystem is given a function dependency child node with a call key that has not been added to the global dependency dictionary.

### Expected Behavior

In the case when there are multiple assets with the same subsystems and dependencies, the dependency function differs in the state var key it uses to access data. That is, if there are two assets with a power subsystem, those two power subsystems have different states, and therefore different state var keys, though the functionality of their dependency functions are identical. In order to distinguish the difference between the two dependency function in the XML file, the call keys should be different because, though they are the same dependency functions, they use a different state var key. An easy way to distinguish call keys is to simply add the asset name as an extension to the call key.

# Subsystem Modeling

The Horizon Simulation Framework is designed with modeling AND simulation in mind. In fact, one could be forgiven for seeing a modeling bias built into the Horizon Simulation Framework. This is not by accident. The development of valid models of the “right” fidelity is critical to conceptual design and requirements verification. Therefore, the Horizon Simulation Framework was designed with an “eye on modeling”.

## The Horizon System Modeling Philosophy

What is the “right” level of fidelity? Well, the answer depends on the maturity of the program in question. Keep in mind that fidelity generally increases with as a program matures. With this in mind, the Horizon Simulation Framework was designed for the system level requirements verification by allowing for maximum flexibility from a modeling perspective. The main algorithm of the Horizon Simulation Framework is really just a time keeper, state verification machine and bookkeeper. The real effort involved in system simulation is incumbent on the engineer working with the Horizon Simulation Framework. The Engineer or Analyst has ultimate control over the exact nature and execution of the models and model development.

The flexibility and extensibility of the Horizon Simulation Framework lies at the heart of the software. The Simulation Framework provides a well-defined set of interfaces between subsystems and the main scheduling algorithm. Therefore, model is king! The Analyst need only focus on developing a model of the “right” fidelity for the simulation. However, the Horizon Simulation Framework does impose a few strict conditions. Namely, no circular dependencies between subsystems are allowed.

## Adding a Subsystem to the Simulation

To help the user design a coherent set of subsystems, the Horizon Simulation Framework provides a set of templates and placeholders for subsystem modeling. If the built in models within HSF are sufficient, attributes of these subsystems can be modified through the Model Inputs XML file. Functionality of these subsystems can also be overwritten with the use of Python scripting as will be discussed within this section. Brand new subsystems can also be added to the system by using Python scripting to overwrite the functionality of the Subsystem Class. The Subsystem Class is an abstract class and defines an interface between the scheduler main algorithm and the set of subsystem models—all the built in subsystems inherit from the Subsystem Class. Every scripted Subsystem modeled by the analyst must be derived from the Subsystem class defined by the Horizon Simulation Framework. The user is free to define any other member variable or functions within a derived scripted subsystem class.

### Subsystem Attributes

The built in subsystems all have customizable attributes that can be modified by simply modifying the XML file. Modifying the XML file is the simplest way to design the system in Horizon. The Required Fields section documents the fields that are needed by all subsystems Each of the built in subsystems’ attributes and their default values are documented below.

#### Required Fields

|  |  |  |
| --- | --- | --- |
| Attribute | Required? | Notes: |
| Type | Yes | The type attribute can be any of the below types or scripted to indicate to the frame work the type of subsystem to initialize. |
| subsystemName | Yes | Name of the subsystem to be used elsewhere in the XML file. |

#### Attitude, determination and Control

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Required?** | **Notes:** |
| slewTime | No | The worst case slew time Default value is 10s |

#### communication

There are no modifiable attributes for the communication subsystem. If a limit is wished to be placed on the downlink data rate a constraint should be used.

#### Earth observing sensor

|  |  |  |
| --- | --- | --- |
| Attribute | Required? | Notes: |
| lowQualityNumPixels | No | The low quality number of pixels indicates to Horizon the number of pixels to use in a low resolution image. The default value is 5000 pixels. |
| midQualityNumPixels | No | The middle quality number of pixels indicates to Horizon the number of pixels to use in a mid resolution image. The default value is 10000 pixels. |
| highQualityNumPixels | No | The high quality number of pixels indicates to Horizon the number of pixels to use in a high resolution image. The default value is 15000 pixels. |
| lowQualityCaptureTime | No | The time required to take a low quality resolution image. The default value is 3 seconds. |
| midQualityCaptureTime | No | The time required to take a mid quality resolution image. The default value is 5 seconds. |
| highQualityCaptureTime | No | The time required to take a low quality resolution image. The default value is 7 seconds. |

#### power

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Required?** | **Notes:** |
| batterySize | No | The size of the battery in W/hr. The default value is 1000000 W/hr. |
| fullSolarPower | No | The amount of power in when the spacecraft is in sunlight in Watts. The default value is 150W. |
| penumbraSolarPanel | No | The amount of power in when the spacecraft is in sunlight in Watts. The default value is 75W. |

#### solid state data recorder

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Required?** | **Notes:** |
| bufferSize | No | The size of the onboard memory storage in bytes. The defualtvalue is 4098 bytes. |

### Subsystem Child Nodes

The child nodes of the subsystems can be used to define initial conditions of the subsystem’s state variables and the dependencies between subsystems. By convention, subsystems can only set the initial state for state var keys that belong to them and dependency child nodes should be written in the subsystem in the direction that is the base of the arrow (i.e. the power subsystem has child dependency nodes of ADCS, COMM, EO Sensor and SSDR).

#### Initial conditions

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Required?** | **Notes:** |
| type | Yes | The type of the HSF Profile associated with the state var key. |
| key | Yes | The state var key to set the initial state of. |
| value | No | The initial value to start the state variable with. Default value is zero. |

#### Dependency

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Required?** | **Notes:** |
| subsystemName | Yes | The size of the onboard memory storage in bytes. The defualtvalue is 4098 bytes. |
| fcnName | No | The function call key to give the subsystem access to its dependent subsystem dependency function. If no dependency function is specified then it is simply an order dependency. |

## Scripted Subsystem Definition

One of the most powerful features of HSF is that it allows the user to implement custom subsystems via Python scripting. This can be done in two ways: deriving from one of the five built in subsystem to modify functionality of one or more functions or by deriving from the abstract subsystem class to implement a brand new subsystem. The process for both methods is identical, though more work must be done when inheriting from the abstract class because none of the methods are implemented.

### Inheriting from a built in subsystem

Before overriding the functionality of a subsystem that is built into the framework, the user must understand the subsystem of interest. The three functions that can be overridden are the CanPerform, CanExtend and DependencyCollector. These three methods have implementations specific to their subsystems and may utilize other methods within the subsystem. New auxiliary methods can be added for use within the CanPerform, CanExtend and DependencyCollector.

All Subsystems

All subsystems inherit from the abstract subsystem and therefor have the same basic properties listed in the table below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Asset | Asset | The asset that the subsystem belongs to. |
| IsEvaluated | Bool | Indicates to the scheduler if the subsystem has been evaluated for the current event. |
| DependentSubsystems | List<Subsystems> | A list of all the subsystems that the subsystem is dependent on. |
| Name | String | The name of the subsystem from the XML file. |
| DefaultSubName | String | The default name given to the subsystem if one is not found in the XML file. |
| IKeys, DKeys, MKeys, BKeys, QKeys | List<StateVarKey> | A list for each type of StateVarKey. The subsystem stores all the keys it has for its states in one or multiple of these lists. |
| \_newState | SystemState | The most recent state, set by the scheduler before the subsystem is evaluated |
| \_task | Task | The current task that the subsystem is being asked to perform. |

**Methods**

The CanExtend and DependencyCollector methods both have default implementations that are used by most subsystems. The default CanExtend method simply checks to see if the time to extend the state to is within the current event, then the method returns true. The default DependencyCollector simply calls all the dependency functions and adds their results together in double HSFProfile. The subsystem’s accessible properties and current CanPerform methods are documented in the sections below.

Attitude Determination and Control

The ADCS subsystem utilized the default CanExtend method and has the properties and methods described below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| POINTVEC\_KEY | StateVarKey<Matrix<double>> | The point vec key is used to store the pointing vector in the state. |
| \_timetoslew | double | The worst case slew time. All slews are assumed to take this long. |

**Methods**

1. CanPerform: The ADCS CanPerform simply checks that there is enough time to slew to the new target before the task ends. If there is enough time to execute the slew, the the pointing vector state is set to the ne pointing vector.

Communication

The COMM subsystem utilized the default CanExtend method and has the properties and methods described below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| DATARATE\_KEY | StateVarKey<double> | The key to access the data rate value of the state to store the data rate that was required to downlink the data. |

**Methods**

1. CanPerform**:** The COMM CanPerform calls its dependency collector to get the data rate required to downlink all the data in the SSDR when the task type is COMM. The COMM CanPerform always returns true, but a constraint on the max data rate may fail later if the data rate achieved was too high for the actual system.

Earth Observing Sensor

The EO Sensor subsystem utilized the default CanExtend method and has the properties and methods described below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| PIXELS\_KEY | StateVarKey<double> | The key to access the number of pixels captured value of the state. |
| INCIDENCE\_KEY | StateVarKey<double> | The key to access the incidence angle of the image captured value of the state. |
| EOON\_KEY | StateVarKey<bool> | The key to access the value of the state that indicates if the sensor is on or off. |
| \_lowQualityPixels | double | The number of pixels captured when a low quality image is takes. |
| \_lowQualityTime | double | The amount of time it takes to capture a low quality image. |
| \_midQualityPixels | double | The number of pixels captured when a medium quality image is takes. |
| \_midQualityTime | double | The amount of time it takes to capture a medium quality image. |
| \_highQualityPixels | double | The number of pixels captured when a high quality image is takes. |
| \_hgihQualityTime | double | The amount of time it takes to capture a high quality image. |

**Methods**

1. CanPerform: The CanPerform method for the EO Sensor determines from the target that is held within the task what type of image to take and then verifies that there is enough time to take the image. If the attempt to take an image is successful, the PIXELS\_KEY is updated with the number of pixels take, the INCIDENCE\_KEY is updated with the incidence angles the image was taken at and the EOON\_KEY is updated to indicate that the EO Sensor was on.

Power

The Power subsystem has the properties and methods described below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| DOD\_KEY | StateVarKey<double> | The key to access the value of the state that records the depth of discharge of the battery |
| POWIN\_KEY | StateVarKey<double> | The key to access the value of the state that records the power into the system from the solar panels. |
| \_batterySize | double | The capacity of the battery in W/hr |
| \_fullSolarPanelPower | double | The amount of Watts entering the system when the spacecraft is in direct sunlight. A constant value for simplicity. |
| \_penumbraSolarPanelPower | double | The amount of Watts entering the system when the spacecraft is in penumbra. |

**Methods**

1. CanPerform: The power subsystem CanPerform collects the power consumed from the other subsystems via the dependency collector then calculates the solar panel power in to update the battery’s depth of discharge.
2. CanExtend: The CanExtend method performs the same function as the CanPerform as the battery DOD is constantly changing as other subsystems have an “idle” power usage and the solar panels are always active.
3. GetSolarPanelPower: GetSolarPanelPower takes in the shadow state and determines the power in depending on the shadow state.
4. CalcSolarPanelPowerProfile: CalcSolarPanelPowerProfile takes in the start and end times, state and dynamic state to determine the shadow state based on the position to call the GetSolarPanelPower function in order to get the power in between start and end time. Once the power is determined at all times, the value of the POWIN\_KEY of the state is updated.

Solid State Data Recorder

The SSDR subsystem utilized the default CanExtend method and has the properties and methods described below.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| DATABUFFERRATIO\_KEY | StateVarKey<double> | The key to access the value of the state to record the data buffer fill ratio (the percentage of memory consumed in the solid state data recorder) |
| \_bufferSize | double | The total amount of memory available in the solid state data recorder in bytes. |

**Methods**

1. CanPerform: Depending on the type of task, the SSDR CanPerform behaves differently. For a COMM task, the SSDR subsystem send as much data as it can to the COMM subsystem in 60 seconds. For the imaging task, the SSDR CanPerform calls its dependency collector to determine how much data it needs to store from the EO Sensor, then updates its state to store it granted the data buffer isn’t full.

## Inheriting From an Abstract Subsystem

At this point, you have added a subsystem to the Horizon Simulation Framework. Lastly, you may need to define any external constraints on the subsystems.

# Subsystem Constraints

A constraint in the Horizon Simulation Framework is an externally imposed constraint on a subsystem. For example, the power subsystem cannot go below 80% State of Charge on the batteries. The value 80% is an externally imposed requirement on the system, not a physical property of batteries. As such, the task of modeling is generally split into two parts; developing a physical model of the system, and imposing any constraints which are derived from a system or subsystem requirement. Constraints can be defined by inheriting from the abstract Constraint class or using the built in constraint called SingleConstraint. Constraints are initialized by using the XML file and the Constraint tag as documented in Chapter 2. A sample of a constraint on the Power subsystems depth of discharge is shown below. It can be seen that the constraint will cause the system to fail is the power subsystem’s depth of discharge exceeds 25%.

<CONSTRAINT

value="0.25"

subsystemName = "Power"

type="FAIL\_IF\_HIGHER"

name="con1">

<STATEVAR type = "Double" key="DepthofDischarge"></STATEVAR>

</CONSTRAINT>

It is imperative that the subsystemName field excatly match the name given to the subsystem in the subsystem’s node. Only the types documented in Table 1 will be accepted by the framework unless a custom constraint type is made. The types on constraints supported by HSF by using SingleConstraint are documented in Table 1.

Table . HSF Single Constraint Types

|  |  |
| --- | --- |
| **Name** | **Description** |
| FAIL\_IF\_HIGHER | When the fail if higher constraint is placed on a state variable, the constraint will cause the system to fail if the state variable exceeds the threshold. |
| FAIL\_IF\_HIGHER\_OR\_EQUAL | When the fail if higher or equal constraint is placed on a state variable, the constraint will cause the system to fail if the state variable exceeds or equals the threshold. |
| FAIL\_IF\_LOWER | When the fail if lower constraint is placed on a state variable, the constraint will cause the system to fail if the state variable falls below the threshold. |
| FAIL\_IF\_LOWER\_OR\_EQUAL | When the fail if lower or equal constraint is placed on a state variable, the constraint will cause the system to fail if the state variable falls below or equals the threshold. |
| FAIL\_IF\_NOT\_EQUAL | When the fail if not equal constraint is placed on a state variable, the constraint will cause the system to fail if the state variable does not equal the threshold. |
| FAIL\_IF\_EQUAL | When the fail if equal constraint is placed on a state variable, the constraint will cause the system to fail if the state variable equals the threshold. |

## Creating a Custom Constraint

The Constraint class is an abstract class that can be inherited in the python The user must simply override the *Accepts()* funtion, similar to how subsystems must override the *CanPerform()* method. The *Accepts()* mehtod takes in a SystemState and returns a boolean of whether or not the constraint passed or failed. Because only subsystems can access their own state from the SystemState, a dependency function should be created so that the constraint can receive the necessary state data to evaluate the constraint. The dependency function should be called from within the *Accepts()* method.

# Putting it All Together

With dependencies, subsystems and constraints defined the user may now create a system model. From start to finish it is suggested that the user diagram the subsystems with their dependencies and how the constraints are applied to the states. From there, the StateVarKeys of each subsystem should be decided along with the names of the dependency functions. Then, the Subsystems can be defined whether by modifying existing subsystems or creating new ones from scratch. The dependency functions should then be implemented if they differ from the ones provided by HSF, followed by the constraints. With the functionality of the model defined, Chapter 3 should be used to create the XML file so that it depicts the model and notifies HSF how to use the dependency functions and constraints. Lastly, the simulation input with the simulation parameters and target deck with the target descriptions should be generated in order to run the simulation.

Chapter

3

Running a Simulation

Input, Runtime and Output

# Input

Three input files to the Horizon Simulation Framework are required; the Model Input File, Simulation Input File and Target Deck Input File. The files and their formats are described below. The Horizon Simulation Framework uses eXtensible Markup Language, or XML, files to store all input data, and the three files are input to the framework as command line arguments. The three files each have required and optional fields as will be discussed in their respective sections.

## The Model Input File

The Model Input File contains all the information required to initialize the model to be simulated. FIGURE is an example listing of a valid Horizon Input File. The following sections will discuss the allowable tags in the Model Input File.

#### Asset tag

The asset is the highest level element within the system. It encapsulates all subsystems with the same dynamic state. At each time step, each asset can be tasked once, so typically subsystems within the same asset are all needed to accomplish a task. The asset combines the system’s motion as defined by the dynamic state, its capabilities as defined by the subsystem, and the constraints on the asset’s state as defined by the constraint.

* **Required:** Yes
* **Attributes:** Asset Name, a unique name for the asset
* **Child Nodes:** Dynamic State, Subsystem (optional), Constraint (optional)

#### Dynamic State tag

The dynamic state tag defines two key elements of the system: the system’s initial conditions of position and velocity, and how the system’s dynamic state should be propagated over time

* **Required:** Yes
* **Attributes:** Dynamic State Type, Initial Conditions
* **Child Nodes:** Equations of Motion (optional)

#### Subsystem tag

The subsystems within the asset can vary drastically in type and interdependency. Each subsystem attribute definition is unique to the scenario and subsystem being simulated. The proper definition of the subsystem parameters and properly setting the parameter value in the subsystem constructor is incumbent on the user. For more on how to add a subsystem, see the section on *Adding a Subsystem to the Simulation*.

* **Required:** At least one subsystem *should* be included in an asset, though it is not required.
* **Attributes:** Subsystem Type, Subsystem Name (all other attributes are variable with subsystem type)
* **Child Nodes:** Initial Condition, Dependency (optional)

#### Contraint tag

* **Required:** No
* **Attributes:** Subsystem Name, Type, Value
* **Child Nodes:** StateVarKey

#### Equations of motion tag

* **Required:** No (default is Horizon Pre-Propagated Orbital EOMs)
* **Attributes:** EOMs type
* **Child Nodes:** None

#### Initial Condition tag

* **Required:** Required when specified as required
* **Attributes:** Type, Key, Value
* **Child Nodes:** None

#### Dependency tag

* **Required:** No
* **Attributes:** Subsystem Name, Dependency Function Call Key (optional)
* **Child Nodes:** None

#### Statevarkey Tag

* **Required:** Required when specified as required
* **Attributes:** Type, Key Name
* **Child Nodes:** None

### The Simulation Input File

#### Scenario Tag

In Figure 2.2.1, the <SCENARIO>…</SCENARIO> tag is the scenario tag. The scenario tag acts as a container for all other tags. The Horizon Simulation Framework looks for the scenario tag as the beginning of the input file.

#### Simulation Tag

The <simulation>…</simulation> tag is shown in Figure 2.2.1. The simulation tag acts to define the parameters which define the overall behavior o the Horizon Simulation. The simulation tag is required and must define the following attributes:

* simStart – The start of the simulation in seconds. For example, in Figure 2.2.1 the simStart attribute is set to 0.0 seconds. Therefore, the simulation will start 0.0 seconds after the JD defined by the startJD attribute.
* simEnd – The end of the simulation in seconds. For example, in Figure 2.2.1, the simEnd attribute is set to 5400.0 seconds or 90 minutes. Therefore the simulation will run for 5400.0 seconds of simulation time after the start of the simulation.
* simStep – The time step the Horizon Simulation Framework takes. In Figure 2.2.1 the simStep is set to 10.0 seconds. Therefore the Horizon Simulation Framework will attempt to schedule tasks every ten seconds.
* startJD – The absolute start of the simulation in Julian Date (JD). The startJD gives the Horizon Simulation Framework an absolute point in time to begin the simulation. All calculations in geocentric inertial coordinates are done with reference to the JD given by the startJD attribute.
* maxSchedules – The maximum number of schedules to maintain at each time step of the simulation. When the number of schedules generated at each time step exceeds the maxSchedules attribute the current set of schedules is sorted and on the best schedules are kept. All other schedules are deleted from memory. At the next time step the Horizon Simulation Framework tries to add events to the reduced set of schedules. In general, the time the Horizon Simulation Framework takes to run increases with the maximum number of schedules maintained. When only one schedule is maintained, the main scheduling algorithm in equivalent to a greedy algorithm. When the maximum number of schedules maintained is very large, the main scheduling algorithm is equivalent to an exhaustive search algorithm.

### The Target Input File

All target information is also stored in an XML file. The name of the target deck file is a command line input to the Horizon executable. Figure 1 shows an example listing of a valid target input file.

<TARGETDECK>

<TARGET

Name = "t4"

Type = "gt"

latitude = "0.0"

longitude = "0.0"

altitude = "0"

Priority = "5"

Value = "5"

MinQuality = "6"

DesiredCapTime = "75600"

NonzeroValCapTime = "32400">

</TARGET>

<TARGET

Name = "t5"

Type = "gt"

latitude = "0.0"

longitude = "120.0"

altitude = "0"

Priority = "6"

Value = "10"

MinQuality = "1"

DesiredCapTime = "57600"

NonzeroValCapTime = "21600">

</TARGET>

<TARGET

Name = "t6"

Type = "gt"

latitude = "0.0"

longitude = "-120.0"

altitude = "0"

Priority = "2"

Value = "4"

MinQuality = "5"

DesiredCapTime = "28800"

NonzeroValCapTime = "25200">

</TARGET>

</TARGETDECK>

Figure . A Sample Target Deck File

Each target deck is a collection of targets. The targets are contained within the <TARGETDECK>…</TARGETDECK> tag. Each <TARGET> … </TARGET> tag defines a target and in general the user can add as many targets as desired. The attributes of each target are customizable by the user and no attributes are required. However, in general the latitude, longitude and altitude attributes should be defined. Likewise the Name and Type attributes are generally defined.

# While Horizon is Running

## The Scheduling Algorithm

The main scheduling algorithm used by the Horizon Framework is discussed below. Simply stated, the Horizon main scheduling algorithm is an exhaustive search algorithm with branch pruning which searches forward in time.

Before the main scheduling algorithm is initiated, all inputs to the scheduling algorithm are constructed from either input files or other information provided by the user to the Horizon Framework. Specifically, the main scheduling algorithm requires the following inputs:

* A system model with an initial state
  + The system model is instantiated prior to calling the main scheduling algorithm and in determined by the XML input file. As described above in more detail, the input file is an XML file which specifies the subsystems, their relevant parameters and any initial conditions.
* A set of tasks
  + As defined in the Horizon Framework Glossary, a Task is an action to be performed at a target, with limitations and suggestions for scheduling. A Target is defined as a named object with a position that may vary. In its most simple form, a Task could be a static location on the surface of the earth with imaging limitations such as minimum elevation angle requirements. Obviously the set of Tasks is application and target type dependent. All information required to generate the set of Tasks is an input to the Horizon Framework and is built before the main algorithm is called.
* The Start Time and End Time of the scheduling period and Time Step Length
  + The main algorithm is time driven and time is monotonically increasing as the main algorithm runs. Therefore, each schedule as a predefined start and end time and a time step must be supplied to the main algorithm. Future versions of the Horizon Framework will have the capability to adaptively determine the next time step based on the current density of tasks which are immediately executable by the system model.

The Output of the Horizon Framework main scheduling algorithm is a set of schedules. The set of schedules represents the best solution to the scheduling problem based on the Value Function assigned to the Horizon Framework. Therefore, the output of the main scheduling algorithm is the optimal solution to the scheduling problem in the sense that the user defined value function is optimized.

The set of proposed schedules returned by the main algorithm are actually a subset of all possible schedules. In general only a small subset of schedules will be returned. The number of returned schedules is a user defined input parameter to the Horizon Framework. When the Horizon output files are generated, only the state data of the best schedule is written to the output file. This keeps the generated data set to a reasonable size. However, if the user would like to use the sub-optimal schedules for analysis, the appropriate data can be generated with only small modifications to the Horizon Framework. The ability to generate several output files, each representing a suboptimal schedule will be included in future versions of the Horizon Simulation Framework.

From a top level view, the functional flow of the Horizon Framework main scheduling algorithm is a set of three loops. The functional form of the Main Algorithm is shown in Figure 4. The outer most loop of the algorithm is the main time loop which drives the whole algorithm. At each point in time, iterate through each previously generated schedule. Next, for each schedule, iterate through all of the tasks in the task set and try to add the task to the current schedule as a new event. One important point to notice is that only the previously generated schedules are considered at each time step. As schedules are generated, they are added to the set of schedules to consider, at the next time step. If not, the scheduling algorithm would enter an infinite time loop at each time step.

Add Empty Schedule

Increment Time Loop

max Schedules

Trim Schedule Set

Increment Schedule

Add New Schedule

Increment Task Loop

time ≥ Current Schedule End Time

can be Scheduled in?

System Can Perform?

All Tasks Considered?

All Schedules Considered?

sim end time?

Return Schedules

1

2

2.a

3

3.a

4

4.a

4.a.i

4.a.i

5

Figure .The flow of the Main Scheduling Algorithm used by the Horizon Simulation Framework.

Let’s look at the main scheduling algorithm in more detail… When the main scheduling algorithm is called, the algorithm is executed as follows:

1. First, add an empty schedule to the set of schedules. The empty schedule acts as a primer to the algorithm. The empty schedule is null and therefore any event can always be added to the empty schedule.
2. Begin Time Loop. For each point in time:
   1. If the current number of schedules being considered is greater than a user defined maximum, determine the value of each schedule and sort the schedules by value. Next, trim the current set of schedules so only the best maximum number of schedules is kept. Finally, delete all other schedules from memory.
3. Begin the Generated Schedule Loop. For each previously generated schedule:
   1. Verify the scheduling time period has not passed for the schedule currently being considered. If the scheduling time period has passed, no more events can be added.
4. Begin the Task Set Loop. For each Task:
   1. If the Task can be scheduled in the currently considered Schedule, create a potential new state and:
      1. If the system can perform the Task create a new event and add a new schedule with the new event to the set of schedules.

5. Return the set of best schedules.

The Horizon Framework main scheduling algorithm is simple in its execution because the job of determining when an event can be added to a schedule is pushed to the system model. During step 4.a.i of the main scheduling algorithm, the system model is asked to verify that adding the proposed task to the schedule is feasible and meets all constraints modeled by the system and subsystems. Therefore, a good system model is critical.

The time driven nature of the main scheduling algorithm does place constraints on how systems and subsystems are modeled. Since the algorithm works forward in time, each previous state is fully determined and cannot be modified by a future state.

## Constraint Checking

When the Horizon Simulation Framework main scheduling algorithm calls the *canPerform()* method of the system class the system and subsystem constraints are verified. If all constraints are satisfied, the *canPerform()* method returns the resulting state of the system which is then added to the schedule currently being considered in the form of an event.

The inputs to the system *canPerform()* method are the final state of the current schedule being considered, the Task which the scheduler is trying to append to the current schedule, and a reference to the final state of the current schedule after the task has been added. As stated above, if the result of the *canPerform()* method is true, the new state of the system is contained in the reference to the new state passed to the *canPerform()* method.

## The Cascade Constraint Checking Sequence

The Horizon Simulation Framework checks constraints, both internal and external to the system models by calling the Cascade Constraint Checking Sequence. Figure 5 shows a visual representation of how the Cascade Constraint Checking Sequence works.

If an Event can be added to the current Schedule, the Subsystem Model returns Pass and the new State Variable.

If an Event cannot be added, the Subsystem Model returns Fail

Subsystem

Model

Pass/Fail

Pass

Fail

Fail

Fail

Previous State

and New Event

Add Event to Sequence

Pass

Figure . The Cascade Constraint Checking Sequence Concept

The Cascade Constraint Checking Sequence works in conjunction with the modeling philosophy employed by the Horizon Framework. In this modeling philosophy, each subsystem model gathers all required information to determine its contribution to the final state of the system from previously validated constraints. In other words, since the state of the system is represented as a set comprised of the subsystem state information, once a subsystems state has been determined, each subsequent subsystem need only query that subsystem for required information. The Cascade Constraint Checking Sequence limits how systems are modeled since once a subsystem state is determined, it can not be modified if the resulting modification would change the state of subsystems whose state has already been determined. In other words, there can be no circular dependencies between subsystem states.

The Cascade Constraint Checking Sequence order is determined by user defined input to the Horizon Framework. The code for ordering the constraints is called prior to the Horizon Framework main scheduling algorithm is called. Therefore, the system and subsystem models and the order in which they’re called must be determined by the user with great care. Internally, the Horizon Framework cannot verify that the state data required by a subsequent subsystem has actually been generated. However, future version of the Horizon Framework will have this capability

The Horizon Framework implements the Cascade Constraint Checking Sequence by assuming there are two distinct types of constraints; those which require information from a subsystem or set of subsystems and those that do not. To account for the two different types of constraints, the system *canPerform()* method generates the new state information by implementing two loops. The first loop checks all constraints which are defined by the user to have other subsystem constraints. The possibility for multiple, single threaded cascade sequences is handled by allowing for multiple lists of constraints. The outer loop of the system *canPerform()* method iterated through this list of constraints. The inner loop then identifies each subsystem in the current constraint cascade, iterates through each subsystem and then calls the *canPerform()* method for each subsystem.

For each constraint, the *accepts()* method of each constraint is called. The *accepts()* method imposes external constraints on the subsystem such as desired minimum state of charge for a power subsystem, or maximum excursion angle for a sensor. External constraints are modeled separately from internal constraints as implied by the Horizon Modeling Philosophy. In general, the internal constraints are physical properties of a system. For example, a battery cannot have a state of charge greater than 100%. An external constraint is an imposed constraint which is generally derived from a system or subsystem requirement. For example, a battery should not have a state of charge less than 80%.

The purpose of separating the two type of constraints is to separate physical models from design requirements. In the future, the Horizon Simulation Framework team hopes to build a utility which can convert a set of requirements into a set of externally imposed requirements. The idea is to automate the process of requirements verification and requirements flow.

After all constraints which require state data from other subsystems have been checked, the system *canPerform()* method calls the subsystem *canPerform()* method for all other subsystems whose state has not been determined. These subsystems are part of the system, but do not require state data from other subsystems nor does their state date feed into any other subsystem constraints or state data calculations.

## Dependencies

As hinted at above, dependencies define the relationship or data required between subsystem models. The dependencies between subsystems must be defined by the user and each dependency function is called by the appropriate subsystem can perform function. Dependency functions follow a convention as defined in Chapter 3.

# Output

The data generated by the Horizon Simulation Framework is stored internally until the end of the simulation or a predetermined pause point is reached. Only at that time is the data generated sent to an output file. While the simulation is running the user does not have access the data being generated.

Currently, only the best schedule is returned as output. Therefore, the schedule data cannot be returned until all possible schedules have been returned. Future versions of the Horizon Simulation Framework will allow the user to select the number of schedules returned as output. This will allow the analyst to perform sensitivity analysis on the set of returned schedules.

## Comma Separated Text Files

All data currently generated by the Horizon Simulation Framework is returned to the user as a Comma Separated Text file. The name of the output file is an input to the simulation framework as part of the main input file. If the user would like to add more output parameters to the output file, the user must modify the main executable in the output section. The user simple looks up the desired state variable by state variable and subsystem name and adds the variable to the output stream using either of the Event Class Methods,

getStateVarAtEventStart (string variableName, string subsystemName), or ,

getStateVarAtEventEnd (string variableName, string subsystemName), or ,.

The user can then open the output file with any text editor or spreadsheet tool to perform analysis or visualization of the Horizon Simulation Framework data.

Glossary

**Constraint** – An externally or internally imposed restriction on the legal values of a variable

**Dependency** – How two or more subsystems are related in state

**Environment** - The entity in which systems reside, and to whose physical rules they are subject

**Event** - A scheduled performance of a Task during a time interval

**Schedule** - A time-ordered sequence of events

**Scheduler** - An actor that creates valid schedules for a system

**Schedule Evaluator** - An actor that assigns values to schedules

**State** - A collection of variable values that record the properties of a system at a specific point in time

**Subsystem** - A component representing an aspect of the system, possibly dependent on other subsystems

**System** - A stateful entity that performs tasks

**Target** - A named object with a position that may vary

**Task** - An action to be performed at a target, whose interaction with the Geometry Model provides limitations and suggestions for scheduling

**Variable** - A varying quantity that completely or partially defines the state of a subsystem

Index

a

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 3, 3

Index 1, 1

Index 1, 1

b

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

c

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

d

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

e

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

g

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

h

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

k

Index 1, 1

L

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

m

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

n

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

r

Index 1, 1

Index 1, 1

s

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

t

Index 1, 1

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

w

Index 1, 1

Index 1, 1

Index 1, 1

Index 2, 2

Index 1, 1

Index 1, 1

Index 1, 1

Inde