“PICASSO” INTERFACE FOR 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

Brian Kirkpatrick

August 2010
Committee Membership

TITLE: “Picasso” Interface for Horizon Simulation Framework

AUTOR: Brian Easton Kirkpatrick

DATE SUBMITTED: January 2010

COMMITTEE CHAIR: Eric Mehiel, Associate Professor and Department Chair

COMMITTEE MEMBER: Kira Abercromby, Assistant Professor

COMMITTEE MEMBER: John Clements, Assistant Professor

COMMITTEE MEMBER: Jordi Puig-Suari, Professor
Abstract

The Horizon Simulation Framework, or HSF, is a modeling and simulation framework compiled from C/C++ source code into a command line program. Picasso is an interface designed to control the input files to Horizon by providing visual tools to create and manipulate the XML files used to define an HSF system of assets, their environment, and other simulation parameters. Picasso also supports the visualization of Horizon output in several different forms, and import mechanics from online space object catalogues.
# Table of Contents

Table of Figures ........................................................................................................................................ vii
Introduction .................................................................................................................................................. 1
  Problem .................................................................................................................................................. 1
  Overview ................................................................................................................................................ 1
  Research and Literature .................................................................................................................. 1
  Review of Alternatives .................................................................................................................... 2
  Background .......................................................................................................................................... 4
Interface Design .......................................................................................................................................... 6
  Deliverables .......................................................................................................................................... 6
  Objectives ............................................................................................................................................ 6
  Requirements ....................................................................................................................................... 6
  Constraints ........................................................................................................................................... 7
Architecture and Program Model ........................................................................................................... 9
  HSF Codebase ...................................................................................................................................... 9
  Picasso Programming Model ............................................................................................................ 10
Implementation ......................................................................................................................................... 12
  Scenario File Definition (HSX) ......................................................................................................... 12
  Input File Management .................................................................................................................... 12
  Output File Management .................................................................................................................... 13
  Serialization ....................................................................................................................................... 13
  Import Tools ........................................................................................................................................ 15
Model Definition and Editing ............................................................................................................... 17
  Scripts ............................................................................................................................................... 17
  Assets ................................................................................................................................................ 18
  Subsystem .......................................................................................................................................... 19
  State Variables ................................................................................................................................... 20
  Dependencies ....................................................................................................................................... 21
  Constraints .......................................................................................................................................... 22
Visualization ............................................................................................................................................ 24
**Table of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A scenario definition file</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Contents of an HSX scenario definition file</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Dependencies and Constraints</td>
<td>23</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Input file structure</td>
<td>26</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Scenario and Environment forms</td>
<td>30</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Assets and subsystems</td>
<td>31</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Subsystem scripting</td>
<td>32</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Target deck</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Visualization</td>
<td>34</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Contents of sample scenario file</td>
<td>41</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Contents of sample environment file</td>
<td>41</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Contents of sample model file</td>
<td>42</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Contents of sample target deck file</td>
<td>43</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Contents of sample output file</td>
<td>44</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Sample subsystem script</td>
<td>45</td>
</tr>
</tbody>
</table>
Introduction

Problem

The Horizon Simulation Framework establishes a common and abstract system for modeling, simulating, and scheduling sets of assets in a common scenario. HSF is cumbersome to use, however, and requires an exacting input format, considerable aerospace and computational knowledge, and an external tool or tools to interpret output in meaningful ways. An intuitive interface is desired that provides efficient graphical means to create and edit HSF input, view and interpret HSF output, and manage / interchange multiple scenarios and models.

Overview

This thesis is presented as a design project. First, the problem is presented with consideration given to existing solutions and relevant literature. Then, design criteria are formalized into a set of guidelines for development. Next, the underlying architecture and model of the programming process are reviewed for both the underlying HSF codebase and the Picasso GUI code. Key features of the project implementation are then described, including the handling of HSF input and output and the critical importance of serialization to this approach. The definition of asset model components is then reviewed and correlated to the interfaces that are used to manipulate the underlying components. In the penultimate chapter, the different mechanisms used to visualize HSF output are described. Lastly, several use cases are presented with the project conclusions and suggestions for future work.

Research and Literature

A great demand for satellite modeling and simulation exists in academic research; however, most projects choose to create their own models. In Connors, Ryu, and Dao’s Modeling and simulation of broadband satellite networks, for example, extensive analysis of satellite data transmission
quality could have been performed without resulting to a project-specific model, which could have saved these researchers time and effort. In the right environment, the sophisticated numerical algorithms used to model data transfer between satellites and ground stations could have been defined using a problem-specific script, allowing these engineers to focus on the core of their problem instead of the model they created.

Similarly, Kang, Sparks, and Banda’s study of *Coordinated Control of Multisatellite Systems* could have benefited from a self-sufficient framework in which only the satellite communications and control systems would require advanced scripting definitions. A common modeling framework like the one sought in this project could facilitate the work of aerospace engineers working on these kinds of projects.

The Horizon framework v2.1, defined by O’Connor, Mehiel, and Butler and outlined in their paper “HORIZON 2.1: A Space System Simulation Framework”, addresses these issues by creating a flexible modeling environment that can be used to generate mission profiles and project system behavior. By creating a tightly integrated environment that uses externally-loaded scripts to maintain flexibility, HSF allows users to model any number and type of assets relevant to their experiment, including customized functionality unique to their project, simply by writing their own scripts.

**Review of Alternatives**

When modeling a given system of satellites, engineers have a number of pre-existing tools at their disposal.

**Simulink**

One such popular tool is Matlab’s Simulink package, which provides a block-based visual interface for defining and arranging system elements and their dependencies in a systems-oriented environment. Simulink includes tools for utilizing transfer functions, a set of built-in system
models, and a suite of output and design tools. However, the construction of a model in Simulink is time-intensive, and the resulting complexity and uniqueness of such a model typically limits its utility and application to the original designer and his or her project. An engineer would also need to write their own set of satellite-based tools, as Simulink is not foremost a satellite modeling program.

**STK**

Another popular satellite modeling tool is the Satellite Toolkit, or STK, produced by Analytical Graphics, Inc. STK is widely used within the professional aerospace community, and comes with a large number of satellite-based tools and resources. However, STK is an expensive and proprietary framework whose complexity places its utility outside the scope of the untrained engineer.

**AutoCAD / Thermal Desktop**

Several toolsets exist as add-ons to the popular design application AutoCAD, most notably the modeling suite Thermal Desktop. While the primary focus of Thermal Desktop is time-marching modeling of satellite system thermodynamics, extensive scripting support and existing tools for the visualization of simulation results make it a potential candidate application. However, like STK, the AutoCAD and Thermal Desktop software suites are large, expensive, proprietary, and possess a steep learning curve.

**Other**

Some engineers may also prefer to create their own project-specific modeling system using a general programming language like C++ or Python. While this has the potential to save engineers a fortune in software acquisition and training costs, it also leads to model system specialization, in which a given model becomes so focused on a particular project that it becomes unrealistic to re-use the same model in future projects. This can be a particularly painful problem for these
styles of hand-coded models, because creating a custom model from scratch can be a monumental undertaking.

**Background**

**Modeling Satellite Systems**

The development of space systems such as satellites and satellite networks tends to be a very expensive proposition, in both time and money. The design and testing of such systems takes place over several years, and programs such as the Space-Based Infrared System (SBIRS) are projected to cost upwards of $10 billion.\(^iv\)

One approach that can reduce the time and expense of satellite development is to focus on efficient and flexible modeling and simulation. An integrated framework that can predict system behavior and state dynamics in minutes or hours of computation time can help decrease development time, predict design problems inherent in preliminary satellite system designs, and provide a means for system testing and prediction that can be reused across multiple programs or even integrated across a diverse selection of assets in a single scenario.

**Horizon Simulation Framework**

The Horizon Simulation Framework (HSF) is a tool that addresses these issues. Written around a flexible optimizer algorithm at a high level of abstraction, HSF is capable of modeling multiple assets in a customizable scenario utilizing built-in functions and external scripts to define system behavior. HSF is a modular, system-level modeling and simulation framework that was written with extensibility in mind. HSF accepts input and provides output in the form of XML files, whose filenames are passed to HSF via the command line executable call.

**Interface Requirements**

In order to use HSF, an interface is desirable. A graphical interface for HSF could provide users with a more effective means to create, edit, process, and translate the results of an HSF scenario.
Users with limited aerospace or computational experience could also use HSF through such an interface, because satellite models and algorithmic details would be hidden underneath a more user-friendly system of graphical controls.

The development of this graphical interface, dubbed “Picasso,” is the subject of this thesis project.
Interface Design

Deliverables

The final product of the Picasso project should include a graphical interface, wrapped around the Horizon Simulation Framework, which provides efficient means to create and edit HSF input, view and interpret HSF output, and manage / interchange multiple scenarios and asset models.

Objectives

The following objectives guide the development decisions of the Picasso project:

- **Usability**: A novice user with basic engineering knowledge should be able to implement and simulate a satellite system model with minimal documentation or required skills. Experienced systems engineers should have a readily-accessible toolset that enables them to control and define all aspects of their system model, its environment, its mission, and the HSF parameters.

- **Extensibility**: The development of the HSF interface should utilize good programming techniques and etiquette in order to ensure that future developers may implement additional features with minimal redevelopment and hassle.

- **Marketability**: The end product should be up to market standards when compared to its concurrent competitors. It should be functional, visible, and intuitive while retaining a powerful toolset to address the complex models and problems encountered by engineers working on real-world problems.

Requirements

The Picasso interface should implement the following features in order to be successful:
• **Script editor:** Because scripted subsystem behaviors are at the core of HSF’s flexibility, a built-in scripting editor should be provided to facilitate the creation and modification of asset models.

• **Target management:** HSF scenarios manage a target deck, or list of targets. The size of the target deck can range from a handful of downlink stations, to hundreds of relevant vehicles and thousands of coverage targets. Picasso should provide an efficient graphical means to manage the locations and properties of a scalable target deck.

• **Scenario export:** In order to pass off scenarios to HSF, Picasso must be able to export all relevant scenario information to the required set of XML files. This should take place seamlessly, without any demands on the user.

• **Horizon execution:** Once a scenario is created and exported, Picasso must be able to call HSF to execute the exported scenario. This should take place within or parallel to the Picasso environment, so the user is not distracted by the need to juggle multiple tasks. While a scenario is being executed, Picasso should not be locked out – the user should be able to continue working.

• **Results visualization:** HSF output consists of files with tables of numbers, typically defining the time history of key state variables. Picasso must provide a means to visualize these results, in graph form or by some variable-specific means, in order to facilitate the analysis and comparison of HSF simulation results.

**Constraints**

In order to be successful, Picasso should avoid:

• **Complexity:** Picasso should be a professionally-developed tool, with a codebase that is sufficiently simple to support future development and extension. Complexity should be
avoided to facilitate the efforts of other programmers to understand and to contribute to the project.

- **Closed-source**: While originally contracted by a private organization, HSF / Picasso are intended as fully usable academic and professional software packages. To promote compatibility, to avoid potential legal issues, and to encourage other developer’s understanding of the Picasso codebase, proprietary source code should be avoided.

- **Redundant features**: Many software projects (especially those subject to multiple generations of development) are subject to an overcrowding of features. Programmers who are not familiar with the pre-existing code should be discouraged from re-creating features by ensuring that the Picasso codebase is readable (well-organized, well-commented, and source-controlled).
Architecture and Program Model

HSF Codebase

Beneath the GUI framework, an independent codebase exists that defines key items of scenario information. This codebase, recognizable by the namespace prefix HSF in the Picasso source code, could theoretically be extended to any HSF project to ensure compatibility. Items defined in this codebase reflect the format of HSF input files that they are used to define. An element in this codebase refers to an item that appears as a unique XML element node in any HSF input file. A component is a key item of scenario information defined by a single input file. The figure below displays the difference between elements and components.

```
<SCENARIO scenarioName="testScenario">
  <SIMULATION_PARAMETERS SimStartJD="2454680.0" SimStart="0.0" SimEnd="15000.0">
    <SIMULATION_PARAMETERS>
  </SIMULATION_PARAMETERS>
  <SCHEDULER_PARAMETERS MaxSchedules="2" CropTo="1" SimStep="30.0">
    <SCHEDULER_PARAMETERS>
  </SCHEDULER_PARAMETERS>
</SCENARIO>
```

Figure 1 - A scenario definition file

Elements

Elements, unlike components, are small items of information that define key properties of scenario items. An element corresponds to a single XML element node in an HSF input file. At the root of an element is serialization and deserialization functionality that allows an input file to be rapidly constructed by iterating serialization over all relevant elements. HSF element types include:

- Asset
- Constraint
- Dependency
- Dependency Function
- Eoms
- IC (initial condition, a declaration of a subsystem state variable)
- Lua
• Lua File
• Simulation Parameters
• Position
• Subsystem
• Schedule Evaluator
• Target
• Scheduler Parameters

Using these elements, every component HSF requires may be constructed.

Components
Components are large-scale elements that define a required segment of an HSF input scenario. There are only four types of components, each corresponding to an HSF input file: Scenario, Model, Environment, and Target Deck. Each of these components is comprised of a set of elements listed above. Using recursive serialization, a component input file can be constructed by writing the serialization results from a component’s child elements to a file.

Picasso Programming Model
Programming models define the common template of objects relationships within a sophisticated program. By constraining the nature of interfaces between families of objects, programming models reduce the complexity of development. Programming models are commonly utilized in GUI programming, where a clear separation between GUI objects and the underlying functional logic simplifies the nature of each object by limiting its scope. One common GUI model, the Model-View-Controller (or MVC) model, was adapted to the Windows Forms framework for the purpose of this project.

Model
In the MVC template, the Model object defines the underlying mechanics of a class of program data. Ideally, the model system is developed independently of the View or Controller objects, which facilitates development across multiple platforms and frameworks. In Picasso, the HSF codebase (all namespaces beginning with HSF) defines Model components using Elements,
Components, and common utilities. Developing the HSF Model codebase separately ensures that the same mechanics are used in any HSF implementation.

**View**

The View objects in the MVC template encapsulate the functionality of GUI objects. View objects have no knowledge of how the information they display is manipulated or processed. One example of a View object might be a drop-down select box that, in Picasso, might allow the user to select a type of target. In the Windows Forms framework, many View objects are already defined, and were easily adapted to Picasso’s needs. If the HSF framework were to be adapted to other platforms, the View objects would need to be rewritten accordingly (for example, if an HSF web interface was created, an HTML select box would be used instead).

**Controller**

Controller objects ‘bind’ the data displayed in View objects with the underlying functional class defined in the Model object. While Model and View objects are blind to each other and to their Controller (that is, they are independent of how they might be controlled), the Controller object processes events to update information in both objects, and to execute specific functionality from the Model object mechanics.

In the Windows Forms framework utilized in Picasso, Controller objects are typically Forms (windows) that contain a number of Controls (Views) and references to their corresponding Model objects. Events such as key-presses or mouse-clicks are processed by the Form (Controller), which then updates the values of View fields and data members of Model objects. Certain events may also be processed by the Controller to trigger specific Model mechanics, such as when a Component is serialized to a file.
Implementation

Scenario File Definition (HSX)

HSX, or Horizon Scenario (eXtensible), is the file format used by Picasso to define a specific scenario. The standard is XML-based, and includes version statistics, author information, and a definition of the root path from which the following scenario file locations are specified:

- **Scenario**: used to define common scenario parameters and propagator information
- **Model**: used to de-serialize a model component and its assets (also, in previous versions, used to store environmental parameters)
- **Target Deck**: used to de-serialize a list of targets

A sample HSX file is shown in figure xx.

```
<hsx path="C:\Coding\Thesis\Picasso v1.0\Sample Data\" author="Brian Kirkpatrick" lastSaved='09:17:13 08-26-2009" version='0.1'">
  <scenario filename="SimulatorInput.xml"/>
  <targetdeck filename="test_targets.xml"/>
  <model filename="AllScripted.xml"/>
</hsx>
```

Figure 2 - Contents of an HSX scenario definition file

Input File Management

The input files used to define an HSF scenario and execute an HSF simulation correspond to Components in the Picasso HSF codebase. Because serialization is built in to every Element and Component, serializing a scenario’s Components produces XML text that is then written to a batch of files in a specified location.

When Picasso launches a particular scenario, all of its components are written to the same directory using unique filenames. These filenames are then passed off to an HSF executable,
whose location may be specified in the Picasso Options window. These files are different from those created and updated by Picasso using the ‘save’ functionality; this allows users to retain and track multiple versions of the same scenario while running that same scenario without effecting the saved files.

**Output File Management**

HSF produces a set of output files in a specific directory structure once a simulation has successfully completed. These files typically contain XML or TXT / tab-delimited time histories of key state variables. When Picasso is notified by the Horizon runner that a particular scenario simulation has completed, the results from these files are loaded into a unique Results node under the respective scenario. From here, a user can view the visualization of output data in several different forms. By avoiding direct interaction between the engineer and the raw HSF output files, Picasso aims to reduce confusion, sensitivity to user error, and effort that might be expended by the user as they attempt to interpret raw data.

**Serialization**

Serialization plays a key role in the HSF codebase and the Picasso interface. Most importantly, serialization allows Picasso to quickly and easily write HSF input files from a set of scenario Components. Several other key Picasso features also utilized the serialization required by every Element and Component.

**Preview**

By serializing a Component into a cache and displaying the resulting XML in an XML browser window, Picasso provides a fast and easy way to preview the input file a particular scenario Component will generate. This helps the user troubleshoot Component problems that may be encountered in HSF execution, and provides a clean under-the-hood peak at how a Component is being formatted.
Open / Save
Serialization allows Picasso to write specific scenario Components to user-specified files. Likewise, deserialization allows Picasso to open previously-saved Components, reading class information directly into HSF objects. The method of serialization / deserialization implemented by all objects in the HSF codebase ensures that, unlike built-in serialization and deserialization packages, XML hierarchy and node types / names will be retained throughout the class structure of all HSF Components and Elements.

Xql Parser
The interpretation of XML data, especially when it must match a format as specific as HSF scenario Components, can be a taxing matter. For Picasso, an XQL (eXtensible Query Language) parser was created that allows developers to rapidly query element attributes, child nodes, and text content from a given XML text block. Implemented using Regular Expressions and lightweight XML readers, the XQL parser responds to SQL-style database queries from a dynamically-structured database (XML).

Undo / Redo
A Memento undo / redo model was implemented in Picasso to improve the ease of the user experience. The Memento model encapsulates changes to objects that are then stored in a pair of Undo and Redo stacks. These changes are defined by storing the string serialized from the changing object, before and after the event took place, and a reference to the object itself. When an event is ‘Undone’, the object reference is called to deserialize the relevant values from the proper string in the Memento Undo stack slice. A similar procedure occurs when an even is ‘Redone’, where the object is deserialized from the proper string in the Redo stack instead.

Cut / Copy / Paste
Object information can be cut, copied, and pasted across multiple forms in Picasso by serializing the object content to the clipboard, then deserializing it to the given location when it is pasted. By
simply storing the serialized string on the clipboard instead of entire object copies, Picasso can even store multiple objects on the clipboard, and then use the XML type information in each string to determine which object should be pasted in a given location.

**Import Tools**

Picasso utilizes several common asset sources to facilitate the collection and use of real-world satellite models in Horizon scenarios. In many scenarios, engineers may benefit from the inclusion of existing assets in the scenario model – for example, including the previous NAVSTAR constellation satellites in a model that attempts to parameterize the flight path of a new addition to the constellation. In doing so, it is useful to be able to reference the current location and propagation of the previously-existing satellites in the NAVSTAR constellation.

Instead of repeatedly entering or updating model and position information for each previously-existing satellite, this information may be referenced from a common asset source. Two sources are supported in the current version of Picasso.

**CelesTrak**

CelesTrak is a webpage-based database that contains two-line elements (TLEs) that define the position and orbit of a given satellite. Users may search for satellites in a given CelesTrak category (‘engineering’, ‘navigation’, etc) and for a specific search term (‘poly’ will return all satellites with a name that includes the string ‘poly’, ‘93’ will return all satellites with an international or NORAD id including the number 93, etc). Users then select the desired satellite from the search results, at which point it is imported into the current Picasso scenario as a new asset with a known position and propagator.

**Poly Open Space Catalog (POSC)**

The Poly Open Space Catalog, or POSC, is a spinoff project from Picasso that collects community observations of space objects, verifies them with other contributors, and uses them to
update the satellite’s best-known position. Users may search for any satellite in the POSC database using a single search term, provided a valid contributor account mail is provided from the correct IP address. (The authentication criteria may be modified in the future.) From the returned search results, the user may select the desired satellite. A new asset is then created using the returned position, orbit, and (if provided) XML-based model information.

More details concerning POSC may be found in Appendix A: POSC.
The definition of asset models to be simulated is the most important part of an HSF scenario. An HSF model consists of a set of physical assets, the scripts that define their functionality, and the parameters that describe their conditions and relationships. Asset state variables are particularly important, because they are recorded and compiled into the data files that constitute the output of an HSF simulation. The following elements define key segments of an HSF model.

**Scripts**

**Definition**

Scripts define the model mechanics that are used to execute the simulation, utilizing the Lua scripting language. Scripted functions most commonly describe the mechanics of specific subsystems, but specific functions for mechanics such as equations of motion can also be specified.

**Format**

Lua scripts are referenced from a model’s LUA element, which includes model scripting parameters (such as whether or not scripting is enabled for the given model). The LUA element contains a set of LUA_FILE children, each of which reference a Lua script file. These files are pulled into the framework, where their functions are parsed and assigned by name. While scripting files can have any name, they are commonly organized by the subsystem whose mechanics they define.

**Interface**

Because scripting subsystem behavior is one of the most important aspects of defining and customizing a scenario model, Picasso implements an embedded scripting editor for each subsystem. Opening a subsystem by double-clicking on the subsystem icon displays this editor.
with an organized tree view of important subsystem scripting information, such as the state
variables, subsystem parameters, and functions that are available in the subsystem’s scope.

To facilitate rapid script development, the editor includes context highlighting, phrase
suggestions (including Lua keywords and managed variables and functions within scope), and
limited syntax enforcement.

**Assets**

**Definition**

An asset is a physical system, at a specific location, that will be modeled by HSF during the
simulation. Assets are composed of multiple subsystems that define the mechanics of different
asset features, such as power systems, communication systems, sensor systems, etc. Asset
locations are updated by the model’s equations of motion, and influenced by relevant subsystems
(such as propulsion devices).

One unique aspect of Picasso assets is that, while they may be customized and built from
scratch, they may also be imported from catalogues of known devices. Two such catalogues
featured in Picasso are CelesTrak and POSC. From these catalogues, an imported asset will have
a known position, COEs, and even (in some cases) predefined mechanics.

**Format**

A model component (xml type MODEL) may contain any number of asset child elements (xml
type ASSET). Each asset must contain an initial position (xml type POSITION) and a reference
to the equations of motion that will be used to update that asset’s position (xml type EOMS).

Each asset element contains a set of subsystem child elements (xml type SUBSYSTEM) that define
the mechanics of its features. State variables may be defined within these subsystem
elements, but their constraints are specified in asset child elements (xml type CONSTRAINT).
Constraints specify a limiting value, the failure condition, and the id of the relevant subsystem.
Assets are further constrained by dependencies between subsystems, which are specified as subsystem child elements.

**Interface**

Within Picasso, assets are managed by a form that features a visual drag-and-drop interface for subsystem editing. Assets parameters – such as initial position, the relevant coordinate system, and equations of motion – are managed by controls at the top of the form, above the subsystem workspace. Subsystems themselves are represented and managed by movable, connectable icons in the subsystem workspace, where they may be easily visualized and managed.

**Subsystem**

**Definition**

Subsystems define specific features of a physical asset. The mechanics of a particular subsystem are typically defined by scripted functions that update the appropriate state variables during the simulation, and may be influenced by their dependencies (a comm system, for example, will be dependent on the available power level). As a result, subsystems and their mechanics are at the heart of a given scenario simulation. It should be noted that subsystems may also be described by a predefined set of mechanics that are hard-coded into the Horizon framework; however, scripting is preferable for the sake of modularity and development.

**Format**

A subsystem element (xml type SUBSYSTEM) contains several key parameters, including its type (scripted vs. predefined), name, id, and the names of key functions. A subsystem may also contain child elements defining state variables and their initial conditions (xml type IC) and their dependencies (xml types DEPENDENCY_FCN and DEPENDENCY, for the scripting function parameters and the dependency itself, respectively).
**Interface**

Subsystems are managed in the asset form, within the subsystem workspace. The subsystem workspace is a blank canvas where the icons that represent subsystems may be created, moved, connected, and edited. Double-clicking a subsystem icon, for example, opens up the scripting editor where subsystem behavior may be edited. Connections, representing dependencies between subsystems, can be created by dragging a line between one subsystem’s independent output (on the right hand side) and another subsystem’s input (on the left hand side). Subsystems are also automatically assigned a random color to differentiate between multiple subsystems with a similar layout.

**State Variables**

**Definition**

State variables are the crux of any simulation. These values represent measurements of interest to the user, or values that are needed to drive each timestep of the simulation. Common examples of state variables include dynamic values (position and velocity), power levels, and data bus rates. Key attributes of subsystem dependencies and constraints are also functions of state variables. Rates of change of state variables may also affect subsystem behavior, particularly when iterative methods are used to simulate asset mechanics.

**Format**

In an asset’s subsystem elements (xml type SUBSYSTEM), state variables are declared by specifying their initial value (xml type IC, for initial condition). Within this element, a state variable’s name, data type, and initial value are specified.

**Interface**

State variables in Picasso are managed by subsystem. When managing a subsystem in the scripting editor, a branch of the tree view control used for organization is dedicated to listing and managing a subsystem’s particular state variables. By adding and removing nodes in this branch,
the user may add and remove state variables. Variable properties (name, type, and initial value) may be edited by manipulating the property grid (also in the subsystem form) once the proper state variable tree node has been selected.

**Dependencies**

**Definition**
A dependency defines the means by which one subsystem’s mechanics may be simulated as a function of variables in other subsystems. For example, a comm system’s behavior may be dependent on how much power is available from the battery system. A dependency access point may be defined in the power system to provide access to the needed value. The dependent feature, in this case the asset’s power system, may then call the specified function to retrieve the desired variable.

**Format**
Dependencies are defined as access points within the *independent* subsystem. When a subsystem is scripted, a dependency is defined by two child elements (xml type DEPENDENCY_FCN and DEPENDENCY) within in the independent subsystem. When a subsystem is not scripted – that is, when it is defined by mechanics hard-coded into the Horizon framework – only the DEPENDENCY element is required.

Each dependency element declares the id of the *dependent* subsystem that will access it. Each dependency function element declares the type and name of the function used as the dependency access point.

**Interface**
Dependencies are managed in Picasso by manipulating the connections between subsystem icons. When a subsystem’s output is connected to another subsystem’s input, the second subsystem becomes dependent on the first. The appropriate access function may then be defined in the first
subsystem, and used in the second subsystem’s scripts. See the figure below for an illustration of the dependency interface.

**Constraints**

**Definition**
A constraint is a threshold for a particular state variable that limits its value during simulation.

**Format**
The CONSTRAINT element defines a particular asset’s constraints, referenced by subsystem id and state variable name. The constraint type (fail if higher, fail if lower, etc) and state variable data type are also specified.

**Interface**
When a subsystem is created in the asset workspace, a [+ ] button is visible in the corner of the icon. This button enables users to add and specify a constraint for a particular subsystem. Multiple constraints can be listed and managed as part of each subsystem icon, as shown below, where a constraint of 12.5 is being placed on the state variable *HeatCapacity*. Note the constraint *BatteryCapacity* that already exists on the Power subsystem.
Figure 3 - Dependencies and Constraints
Visualization

Once a scenario in Picasso has been executed, the simulation runs in a Horizon executable on a separate process. When the process has completed, the runner in Picasso acknowledges the simulation and loads the resulting output data into Picasso’s Results visualization forms.

Visualization of output data is an important part of Picasso’s functionality. A scenario simulation is not very useful if the user cannot view the results.

Output Data

Horizon simulations write time histories of state variables into raw TXT and XML files. These files are labeled and organized by asset and subsystem, but their content is difficult to read and process. Therefore, Picasso contains a number of mechanisms by which simulation output data may be viewed and compared.

State Histories

A simple time history plot of state variables is the fundamental workhorse of simulation output analysis. In Picasso’s State History form, multiple plots may be viewed and compared simultaneously. They may also be hidden to customize a particular view, and their interpolation (linear vs. cardinal) may be toggled. Specific time history values are displayed simply by moving the cursor over any plot to the desired moment in the time history; so long as the cursor is over a plot, the values for each of the visible plots are displayed for that moment.

Model View

Sometimes, it is useful to see a 3d view of asset positions and movement over time, especially with respect to the earth and each other. Picasso implements an OpenGL context, rewritten in C#, to render and animate key assets and the earth, according to the asset positions recorded in the simulation output. Mouse-based camera movements allow the user to rotate about the earth and
zoom in on particular features. With sufficiently well-defined meshes and textures, the model view may also be used to record video or screenshots of a particular simulation for presentation or marketing purposes.

**Ground Tracking**

When assets are orbiting about the earth, it may be useful to know where they are in relation to the earth’s surface and the scenario’s target deck. The Ground Tracking form in Picasso animates each asset and (if relevant) their coverage area over the time history of the simulation, on top of a projection of the earth’s surface and relevant target locations.
Use Cases and Conclusion

First Use Case

In the first use case, we would like to demonstrate the ability of Picasso to load, simulate, and view pre-existing models and large target decks. We will be using the Aeolus scenario, which has been used as a testing standard for Horizon in past development. This scenario set is publicly available at http://hythos.net/horizon/Case0_input.zip, and consists of the following files:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>scripts</td>
<td></td>
<td>File Folder</td>
</tr>
<tr>
<td>Access.txt</td>
<td>1 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>Adcs.txt</td>
<td>1 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>AeolusDependencies.txt</td>
<td>2 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>Comm.txt</td>
<td>1 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>EOSensor.txt</td>
<td>3 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>Power.txt</td>
<td>4 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>ScheduleEvaluator.txt</td>
<td>1 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>Ssdr.txt</td>
<td>2 KB</td>
<td>Text Document</td>
</tr>
<tr>
<td>Aeolus.hsx</td>
<td>1 KB</td>
<td>HSX File</td>
</tr>
<tr>
<td>Aeolus_model.xml</td>
<td>4 KB</td>
<td>Safari Document</td>
</tr>
<tr>
<td>Aeolus_scenario.xml</td>
<td>1 KB</td>
<td>Safari Document</td>
</tr>
<tr>
<td>Aeolus_targetdeck.xml</td>
<td>76 KB</td>
<td>Safari Document</td>
</tr>
</tbody>
</table>

Figure 4 - Input file structure

Note that, while this is a common structure for Horizon input, there are no requirements to place HSX input files in any organized fashion, because the specific component and scripting file locations are specified in the HSX and model files themselves, respectively.

We load these files into Picasso by simply opening the main scenario file, Aeolus.hsx. Doing so iteratively opens the component files (model, scenario, and target deck definitions) specified within Aeolus.hsx to create a scenario entry in the Picasso workspace.

Once the scenario is open, we may browse to, for example, the target deck by selecting the “Target Deck” node from the tree view. The form that appears shows a map of the earth’s surface with target icons at their respective locations. In this case, there are around 275 different
targets in the target deck; use the zoom and pan (click and drag) functionality to get a better look at where specific targets are located; double-clicking on their icon will open the form for editing the target itself. You may also change the target location by dragging the icon around the map.

Other components may be viewed and managed in the same way. Clicking on the *Scenario* node, for example, opens a form where scenario properties (including simulation parameters) may be tweaked.

Once scenario is ready to be executed, make sure the proper settings have been entered. (These include parameters like the HSF runner location, which will likely vary from computer to computer.) Then, with the desired scenario selected, choose *Project -> Launch*. A runner will appear, asynchronously tracking the progress of the Horizon simulator and reporting status updates within the form. (You can continue to work in Picasso while a scenario is being simulated; the simulation runs off of a backup copy of the scenario files that are unaffected by changes to regular saved scenario files.)

Once a simulation is complete, the results may be viewed. If the simulation was successful, a message will report its completion and the ‘Results’ node in that scenario’s tree view will be enabled. Three options exist for viewing visualizations of the Horizon results:

- **Under State Histories**, a user may customize the time-history plots of state variable values produced from a simulation. Specific plots may be loaded and organized from the toolbar in the *State Histories* form. Each specific plot may be managed via the icons in its upper-right corner. These options include interpolation settings (linear vs. spline) and an option to ‘hide’ the plot (listing it in the left-hand side and removing it from display). Multiple plots may be displayed and compared on the same simulation timescale.

- **Visualization** allows a user to visually compare the location, orientation (if provided), and mesh / model (if provided) of the assets in a scenario with respect to the earth and each
other. This is accomplished by means of an OpenGL rendering context that displays a rotating earth and satellites with positions and orientations generated from the simulation. Because OpenGL rendering is easily extensible (particularly with the use of externally-defined models and shaders), this view can also be used to create market-quality video renders of asset behavior.

- The ever-popular Ground Overlay view is a mainstay of aerospace culture, and would be sorely missed if not implemented. This visualization animates satellite locations as projected onto the surface of the earth.

**Second Use Case**

The second use case will demonstrate the ability to create an entire scenario from scratch, using the built in tools Picasso features.

Creating a new scenario in Picasso is easy: simply select File -> New or click the ‘New’ icon in the toolbar. After Picasso asks you for the new scenario’s name, a blank scenario is created with all of the necessary components. The new scenario and its components appear in the tree view on the upper-left side of the window, allowing instant navigation to any part of the scenario. For example, click ‘Scenario’ node to open the form that exposes scenario settings and parameters.

Once a scenario is created, each component may be managed in the same manner as the first use case. In this example, we will looker closer at the Asset and Subsystem tools. Right-clicking on the Model node of a new scenario displays a menu with an Add Asset option. This creates an empty asset and displays the form. Asset properties, such as initial location and propagation methods, may be specified here. Most importantly, the workspace below may be used to create and manage asset subsystems and their relationships.
Create several subsystems by double-clicking in the workspace of the asset form. Each icon is assigned a random color and a name you may specify. Clicking on the ‘input’, or left-hand node, of an icon allows you to specify a subsystem as dependent on another (click another subsystem’s ‘output’, or right-hand node, to complete the dependency). You can also specify constraints on a subsystem by clicking the ‘[+]’ button in the subsystem icon’s corner. Each constraint will be listed below the icon once created; clicking the constraint again will remove it.

Double-clicking a subsystem icon displays the meat-and-potatoes of Picasso’s power, the scripting editor. Each subsystem must reference several entry points in its scripts to be properly utilized: initialization, extension, and utilization. Additional functions may be added, and referenced from each other or any of these entry points. State variables and parameters, may also be specified here using the node interface in the top-left corner of the form.

Scripts themselves are written in the Lua scripting language. An interpreter is maintained by the Picasso program to validate your scripts as you write them; validation and save status information is displayed in the form of a red-yellow-green control at the top of the editor, and detailed status messages appear at the bottom. Additional editing tools include the symbol suggestion box, which appears when you begin typing and can be used to rapidly produce code by auto-completion, and the parameters suggestion box, which appears when a function call is written to remind the author of the proper function header.

Results

The results of the Picasso project are shown in the following screenshots, each accompanied with detailed explanations.
A new scenario has been created in the figure above, and the scenario’s parameters and environmental settings are open. While these forms are relatively small and minor, they defined important parameters and assumptions of the model that will be simulated. Note that, while there are multiple scenarios open, only one scenario’s forms are being displayed at a time.
Subsystems are being created and managed in the figure shown above. Several subsystems, such as the Processor and Antennae, display constraints that have been defined. The Processor has also declared a dependency on the Power subsystem, as seen by the line connecting Power output to Processor input. Lastly, note the asset fields at the top of the form. The asset initial position and coordinate system are managed on the left, while the equations of motion used to propagate the asset position are specified on the right.
Figure 7 - Subsystem scripting

This screenshot depicts a Processor subsystem that is being defined by the user. The base required functions are being written, and several state variables have been defined. Note the symbol suggestion box and symbol highlighting features, intended to increase the efficiency of script development.
Figure 8 – Target deck

The target deck and target form for a new scenario are shown above. Several targets have been created, and one target in particular (on the southern coast of Portugal) is being edited. The target deck form includes features for panning and zooming, as well as the ability to create targets directly by double-clicking.
Several windows for scenario visualization are shown in the figure above. Time history plots are being viewed in the upper-left comparing several state variables. On the right, the visualization of the satellites’ orbits are being rendered, while a ground overlay animates their positions with respect to the earth’s surface on the bottom-left.

**Future Work**

While the final version of Picasso is fully capable of creating and managing HSF scenarios, there are several areas where additional work may improve Picasso’s polish and extend various
features. These areas are prime candidates for future students to continue work on Picasso in senior- and Masters-level projects.

**Visualization Polish**
The code that implements Picasso’s ability to visualize results is rooted in OpenGL rendering calls, and is hence highly extensible. Future work could be done improving the quality of visualization rendering using such tools as multi-texturing, satellite- and planet-specific shaders, improved camera maneuverability, and stronger support for screen and video capturing.

**Subsystem-Specific Forms and Controls**
Certain subsystems, or families of subsystems, are very common in satellite modeling and would benefit from manipulation via specialized forms and controls. For example, a unique form could be created to manage a particularly popular power system or propulsion system. Creating specialized forms for subsystems in this manner can increase the efficiency of the user’s modeling efforts by reducing the redundancy of particular simulation definitions while exposing unique key variables to direct user manipulation.

**Feature-Rich Mapping Support**
The current mapping approach supports a single high-resolution texture that renders a projection of the earth’s surface. Future work could be performed to support more scalable texture approaches, such as those implemented by a Google Maps, NASA World Wind, or Bing Maps API. With some of these APIs, a detailed 3d view of surface feature is also possible. Future work could also be performed to extend mapping support to multiple projections.

**Continued POSC Development**
While POSC has the potential to be a very powerful aerospace community tool, the database volume and algorithm sophistication are currently very limited. Future work could focus on increasing the accuracy of POSC position estimates, calculating reasonable margins of error,
supporting multiple element orbital projection algorithms, and implementing POSC routines required to support and update community contributions.

**Implement / Extend HSF Standardized Codebase**

Picasso utilizes a standardized HSF codebase that remains independent of GUI functionality while defining and supporting the underlying structure of HSF scenario definitions and mechanics. This codebase can be used as a standardized library for HSF modeling of the Horizon simulator itself, guaranteeing compatibility between Picasso and HSF models. It can also be used to extend support of HSF scenarios to third-party APIs by providing a single standard for model interaction.

**Conclusion**

Picasso creates an easy-to-use and intuitive interface to the Horizon Simulation Framework. By wrapping command-line functionality and the interpretation of results in a modular GUI, Picasso enables a far greater range of engineers to utilize the modeling power of HSF. This satisfies the *usability* requirement of the software design specifications.

By including development tools such as a full-featured scripting editor and a graphical asset system design workspace, Picasso also increases the speed and fidelity with which models can be produced. Development efficiency is also facilitated by features such as undo / redo and copy / cut / paste abilities. This means that Picasso satisfies the *marketability* of Picasso’s design specifications, because it ensures an efficient use of time and resources that equals or surpasses similar products.

Lastly, by including novel import and export tools, utilizing a standardized component- and element-based codebase, and by following good GUI programming procedures, Picasso maintains a modular and flexible framework of code that can be used for multiple projects and
easily improved in the future. This means that Picasso satisfies the \textit{extensibility} of the design specifications.
Appendices

Appendix A – POSC

Background
The Poly Open Space Catalog, or POSC, is an effort to provide a community-updates database of COEs for objects orbiting the Earth. By establishing an open catalog of space objects, POSC attempts to simplify orbit modeling for third-party user tools.

Contributions
Users whose observation platforms have been verified may contribute observations to the POSC via the POSC API. Observations consist of a set of measurements (the position and velocity of the target object), the identity of the object, the time at which the measurement was made, and the uncertainty associated with the platform's observation.

Once a contribution is made, it is not used to update the position of the object until it has been independently verified by two other POSC users. To facilitate verification, POSC users may enable their platform to respond automatically to queries from the POSC API, tracking and recording the requested target position and velocity.

Queries
The purpose of POSC is to provide a dynamically updated space catalog that may be used by third-party applications. The POSC API supports queries made to the POSC databases, returning COEs or position / velocity estimates that are calculated from recent object observations.

The requested COEs, as well as an estimate of their uncertainties and the time since the last observation update, may be returned and parsed in several formats. The fastest format is in a CSV file queried via an HTML page request. A more flexible format is an XML stream also queried via an HTML page request. Both formats support the query of multiple object information sets.
Additional information may be queried via a separate XML request that details known parameters of a given satellite. This format, however, is not universal (it is meant to be used to import an Asset object in the Horizon Simulation Framework), but the open nature of POSC allows this feature to be used by any other software package, as well.

Given multiple queries, the API estimates an object position by several potential methods. Each method will be implemented in order of increasing complexity:

- Returns the most recent observation
- Returns the most recent observation, projected into the future
- Returns a weighted average of projected locations from each observation
- Utilizes weighted least squares to estimate object position based on past observations and their uncertainties
- Kalman filter weighs uncertainties of each past measurement projection and returns best current guess, with uncertainties

Database

POSC stores three different entries of information:

- A list of contributors and the IP at which observation requests may be made
- A list of observations made by POSC contributors, including uncertainties, times, and whether or not the observations have been verified
- A list of the objects in the POSC observation table, and details concerning their systems

Queries to POSC for a particular object in the catalog first verify the user by comparing authentication credentials with the table of contributors. Then, the object information is pulled
from the object table. Lastly, the best estimate of the object’s position is calculated from recent observations in the observations table, merged with object information, and returned to the user as a formatted XML stream.
Appendix B – Sample Input File

The following sections contain sample input files for each given component.

Scenario

The scenario component contains simulation and scheduling parameters, as well as the scenario name and values defining the simulation time scope and step. For example, the simulation is scheduled to start at $t=0$ and end at $t=1$, etc.

```xml
- <SCENARIO scenarioName="Unknown Scenario">
  <SIMULATION_PARAMETERS SimStartJD="0" SimStart="0" SimEnd="1"/>
  <SCHEDULER_PARAMETERS MaxSchedules="1" CropTo="1" SimStep="10"/>
</SCENARIO>
```

Figure 10 - Contents of sample scenario file

Environment

The environment component defines assumptions and environmental parameters. While currently limited to sun vector assumptions, the environment component could also contain information defining the desired earth J2 propagation model, whether or not lunar effects are included in the simulation, etc.

```xml
- <ENVIRONMENT>
  <SUN isSunVecConstant="True"/>
</ENVIRONMENT>
```

Figure 11 - Contents of sample environment file

Model

The model file contains all information defining a scenario’s assets, constrains, dependencies, and scripting sources. Model files contain a great deal of information; the sample shown below outlines a single-asset model with a handful of heavily-scripted subsystems. (For brevity’s sake, some subsystems and constraints are minimized.)
Target Deck

The target deck file contains a list of targets, their positions, and parameters defining how each target will be integrated into the simulation (as a vehicle, a comm station, a location to be viewed, etc).
Figure 13 - Contents of sample target deck file
Appendix C – Sample Output File

HSF output files consist of an access report for each target and time-history values for the key states of each asset subsystem. The output file shown below records the number of pixels captured by a satellite sensor at different points in time.

```xml
<Base>
  <SubsystemData name="samplePixels" type="numeric">
    <AssetData assetName="Asset1">
      <Data data="0" time="0"/></Data>
      <Data data="5000" time="31.5"/></Data>
      <Data data="0" time="32.5"/></Data>
      <Data data="5000" time="61.5"/></Data>
      <Data data="0" time="62.5"/></Data>
      <Data data="5000" time="91.5"/></Data>
      <Data data="0" time="92.5"/></Data>
      <Data data="5000" time="121.5"/></Data>
      <Data data="0" time="122.5"/></Data>
      <Data data="5000" time="151.5"/></Data>
      <Data data="0" time="152.5"/></Data>
      <Data data="5000" time="181.5"/></Data>
      <Data data="0" time="182.5"/></Data>
      <Data data="5000" time="211.5"/></Data>
      <Data data="0" time="212.5"/></Data>
      <Data data="5000" time="241.5"/></Data>
    </AssetData>
  </SubsystemData>
  + <SubsystemData name="IncidenceAngle" type="numeric"/></SubsystemData>
  + <SubsystemData name="EOSensorOn" type="numeric"/></SubsystemData>
  + <SubsystemData name="DataBufferFillRatio" type="numeric"/></SubsystemData>
  + <SubsystemData name="DataRate(MB/s)" type="numeric"/></SubsystemData>
  + <SubsystemData name="DepthofDischarge" type="numeric"/></SubsystemData>
  + <SubsystemData name="SolarPanelPowerIn" type="numeric"/></SubsystemData>
</Base>

Figure 14 - Contents of sample output file
Appendix D – Sample Subsystem Script

The following subsystem script defines satellite comm system mechanics. *Comm_init* is used to initialize the system; *Comm_canPerform* reports if a given task may be performed; and *Comm_canExtend* reports if a state event may be extended to the end of evaluation time (and adjusts the timing accordingly). While only these required functions are present in this subsystem, any number of other supporting functions may also be included and used within these entry points.

```plaintext
function Comm_init(sub, subParams)
    sub:addKey(dStateVarKey('DataRate(NB/s)'))
end

function Comm_canPerform(oldState, newState, task, position, environment, deps, subParams)
    if (task.GetType() == "CommTask") then
        newProf = deps:callDoubleDependency('COMM_SUB_getDataRateProfile')
        if (not newProf:empty()) then
            newState:setProfile(dStateVarKey('DataRate(NB/s)'), newProf)
        end
    end
    return true
end

function Comm_canExtend(state, position, environment, evalToTime, deps, subParams)
    if (state:getEventEnd() < evalToTime) then
        state:setStateEnd(evalToTime)
    end
    return true
end
```

Figure 15 - Sample subsystem script
Appendix E – Terminology

Asset – An HSF asset is a physical object in a specific location whose mechanics are modeled by the simulation. Asset mechanics are defined by multiple subsystems, whose behaviors are determined by either scripts or predefined mechanics operated with given parameters.

Component – A fundamental element of an HSF scenario definition, encapsulated in a separate file. Examples include ENVIRONMENT, SCENARIO, etc.

Element – An XML element in an HSF scenario component. Examples include POSITION, ASSET, or TARGET. Components are composed of multiple elements.

Model – In HSF, a Model consists of a collection of physical Assets and the parameters that govern their propagation and relationships.

Scenario – A scenario is a single HSF use case, containing all relevant information needed to run an HSF simulation.

Subsystem – A subsystem encapsulates a feature of a particular asset, such as communication or power items. Subsystem mechanics are modeled during the simulation by scripts or predefined methods.

Target – A target is an object whose mechanics are not simulated in the scenario, but whose position and type are still relevant to the model. A target may consist of a communication station, a vehicle, etc.

Target deck – A collection of all targets utilized by a given scenario.
Appendix F – Components and Elements Reference

ASSET – A physical system to be propagated within the simulation. Assets are composed of multiple subsystems, as well as the constraints and dependencies that compose their relationships. Each asset is defined within the MODEL component.

CONSTRAINT – A constraint limits the value of a particular state variable. Constraints are specified by type (fail if higher, low, etc), value threshold, subsystem id, and variable type / name. Subsystems are defined within their respective ASSET elements.

DEPENDENCY – A dependency element indicates that another subsystem is dependent on a property of the subsystem where the dependency is defined. Dependencies are specified by subsystem id, and (if scripted) the access function used to retrieve the relevant value. Dependencies are defined within the independent subsystem.

DEPENDENCY_FCN – This element specifies scripting parameters, such as function name and type, for an access function which specifies a dependent value. Dependency function elements only exist for scripted dependencies, and like their corresponding DEPENDENCY elements, are defined within the independent subsystem.

ENVIRONMENT – An environment component wraps assumptions and parameters defining the environment in which the simulation takes place.

EOMS – An EOMS element typically appears as a child of a POSITION element, and defined the equations of motion used to propagate an asset position.

IC – Initial conditions, when declared, indicate the starting value and type of a state variable. IC elements are defined within their relevant SUBSYSTEMS. Note that an IC element is different from an ICs attribute, which can appear in a POSITION element, but instead defines an asset’s starting position.
LUA – A LUA element, a child of a MODEL component, contains a series of LUA_FILE elements that define where relevant script files may be found. The LUA element itself does not contain any information, save a Boolean flag that indicates whether or not scripting is enabled for that particular model.

LUA_FILE – This element indicates a scripting file that should be loaded at simulation time; the script will contain certain functions that define subsystem mechanics, equations of motion propagation schemes, etc. Each LUA_FILE element is a child of the MODEL component’s LUA element.

MODEL – A MODEL component defines a scenario’s simulated systems and their mechanics. Child elements of the MODEL component include the LUA element, the SCHEDULE_EVALUATOR element, and a set of ASSET components.

POSITION – Asset and target positions are defined in a POSITION element. This element contains initial conditions of a location, the coordinate system used, and (in the case of mobile assets) the EOMS element that defines how the position will be updated.

SCENARIO – The SCENARIO component contains the name of a given scenario, and SIMULATION_PARAMETERS and SCHEDULE_PARAMETERS elements that define fundamental simulation values.

SCHEDULE_EVALUATOR – This element defines, if a model is scripted, the entry point for the model’s schedule evaluator by function name. The SCHEDULE_EVALUATOR element is a child of the MODEL component.

SIMULATION_PARAMETERS – Start, stop, and reference times are defined in the SIMULATION_PARAMETERS element, which is a child of a scenario’s SCENARIO component.
STATEVAR – The STATEVARS element appears as a child of a CONSTRAINT element; it specifies the name and type of the state variable that is being constrained.

SUBSYSTEM – Each SUBSYSTEM element appears as a child of a model’s ASSET elements. SUBSYSTEM elements define the mechanics of a particular asset system, either by scripted or predefined parameters. In addition to the name, parameter values, and id number of the relevant subsystem, a SUBSYSTEM element contains IC elements and DEPENDENCY (with DEPENDENCY_FCN, if scripted) elements specifying the subsystem’s state variables and dependency access points.

SCHEDULER_PARAMETERS – This element defines parameters for the scheduler, such as cropping scale, simulation step size, and the maximum number of schedules.

SUN – The SUN element, a child of the ENVIRONMENT component, defines properties of solar effects used in the simulation. This can include, for example, whether sun vectors are treated as constant.

TARGET – A TARGET element defines a particular simulation target, including its task type, name, target type, etc. Each TARGET element also contains a POSITION element defining its initial position and the coordinate system used.

TARGETDECK – This component contains a list of TARGETDECK elements defining a scenario’s various targets. No other information is included.
Bibliography


Referenced Literature (Endnotes)


