Multi-stakeholder Interactive Simulation for Federated Satellite Systems

Paul T. Grogan
Engineering Systems Division
Massachusetts Institute of Technology
Cambridge, MA 02139
ptgrogan@mit.edu

Alessandro Golkar
Skolkovo Institute of Science and Technology
Skolkovo, Russia
golkar@skolkovotech.ru

Seiko Shirasaka
Graduate School of
System Design and Management
Keio University
Yokohama, Kanagawa 223-8526 Japan
shirasaka@sdm.keio.ac.jp

Olivier L. de Weck
Department of Aeronautics and Astronautics
and Engineering Systems Division
Massachusetts Institute of Technology
Cambridge, MA 02139
deweck@mit.edu

Abstract— Federated satellite systems (FSS) are a new class of space-based systems which emphasize a distributed architecture. New information exchanging functions among FSS members enable data transportation, storage, and processing as on-orbit services. As a system-of-systems, however there are significant technical and social barriers to designing a FSS. To mitigate these challenges, this paper develops a multi-stakeholder interactive simulation for use in future design activities. An FSS simulation interface is defined using the High Level Architecture to include orbital and surface assets and associated transmitters, receivers, and signals for communication. Sample simulators (federates) using World Wind and Orekit open source libraries are applied in a prototype simulation (federation). The application case studies a conceptual FSS using the International Space Station (ISS) as a service platform to serve Earth-observing customers in sun-synchronous orbits (SSO). Results identify emergent effects between FSS members including favorable ISS power conditions and potential service bottlenecks to serving SSO customers.

TABLE OF CONTENTS

1. INTRODUCTION

Federated satellite systems (FSS) are a new class of space-based systems emphasizing a distributed architecture [1]. They contrast with most existing space-based systems which are monolithic (single spacecraft) or distributed but centrally-managed (constellations). Resource exchanging behaviors during operations enable FSS to leverage latent capabilities and comparative advantages of member satellites. Initially,

information and currency exchange establish a business case for FSS participation and future extensions may include wireless power exchange. Information exchange provides new services such as inter-satellite links to transport, store, or process information, potentially reducing on-board communications, memory, or processing requirements for customers.

Architectural complexity is dominant in FSS and the coupling of social and technical dimensions creates a challenge for planning activities. On the technical dimension, constituent systems operate independently as complex systems interacting with other complex systems. On the social dimension, independent stakeholders control constituent systems and decisions are driven by local, potentially-conflicting objectives. Both dimensions are interlinked where focusing on only one dimension can limit validity for the other. For example, economic models may adequately capture profit-seeking behavior but miss implications of design feasibility. Similarly, a purely technical model may establish a feasible system design but fail to attain market viability by misjudging differing stakeholder values or requirements. This paper presents preliminary work to define a prototype simulator for interactive stakeholder evaluation of FSS concepts to gain insights early in the architecture and design process. Future extensions may apply these tools in collaborative design sessions where independent stakeholders coordinate objectives and interactions for a future FSS.

This paper is organized as follows. First, a functional classification of FSS identifies new functions compared to traditional satellite systems and the dual challenges of FSS integration and collaboration are described in the context of sources of complexity and system-of-systems engineering. Next, a FSS simulation architecture is defined using the High Level Architecture (HLA). Two sample federates are presented using the simulation architecture to control space- and ground-based assets and provide 3D visualization. Finally, a prototype application uses the International Space Station (ISS) as a host platform for FSS supplier hardware for customers in sun-synchronous orbit. The paper concludes with a summary of results and future work.

FSS Functional Classification

A functional classification helps to understand the architectural implications of FSS. Table 1 uses a 5×5 framework from [2] which includes five functions (transforming, trans-

---

Table 1. **Functional Classification of Satellite Systems**

	Matter	Energy	Information	Currency	Organisms
Transforming	Propulsion unit	Photovoltaic panel, Electronics, Flywheel, Gyroscope	Processor, Sensor	—	—
Transporting	—	†	Radio antenna, Laser diode	—	—
Storing	Propellant tank	Battery, Flywheel, Gyroscope	Memory	—	—
Exchanging	—	†	⋆	⋆	—
Controlling	Command and control	Command and control	Command and control	—	—

⋆ Immediate new functions for FSS members   † Future new functions for FSS members

porting, storing, exchanging[2], and controlling) operating on five types of operands (matter, energy, information, currency, and organisms). This framework is believed to be complete for classifying engineering systems.

Existing satellite functions operate on propellant (matter), electrical energy, angular kinetic energy, and information. Tanks store propellant which is transformed for orbital maneuvers. Batteries store electrical energy which is transformed by photovoltaic panels (generation) and on board electronics (consumption). Flywheels and gyroscopes store angular kinetic energy which is transformed for attitude control. Information is stored in memory, transformed using sensors (generation) and processors, and transported with communications equipment such as radio antennas and laser diodes. Control of matter, energy, and information functions is exerted by command and control components. While customers of some satellites exchange currency as payment for services rendered, this can be considered a derived function rather than the satellite independently generating revenue.

Resource exchanging functions are the primary addition of FSS. An initial FSS concept adds information exchange which allows data links to transport information across system boundaries. Once exchanged, the FSS members can transform, transport, or store the data using existing components. As a second example, future innovations may allow wireless exchange of energy across system boundaries using novel transport technologies. Once received, the energy can be transformed or stored for future use. In both cases, there is strong incentive for market conditions to establish competitive prices for currency exchange in compensation for the exchanged resources or services rendered by a FSS member.

FSS Integration and Collaboration

The distributed architecture of FSS allowing exchanging functions contributes additional design complexity, here understood as a "measure of uncertainty in achieving the specified [functional requirements]" [3]. This uncertainty arises from technical and social dimensions of integration and collaboration respectively.

Three types of technical complexity arise in FSS. First, each FSS member is a complex system in itself. Second, resource exchanging functions introduce interactions between each pair of FSS members. Third, emergent effects of compositions of pairwise interactions impact the FSS as a whole. Similar issues arise in satellite constellations, however FSS differ in their structure as a system-of-systems (SoS) which

introduces additional social complexity.

A SoS exhibits independent operational and managerial control over its components [4]. A satellite constellation, for example, is not considered a SoS because a central authority controls the member spacecraft. Constituent systems in a SoS may join or leave depending on localized value judgments and there is no centralized authority to force actions or design decisions.

Viewing FSS as a SoS contributes two key challenges. First, each FSS member is controlled by an independent entity, likely introducing barriers to collaboration. During the planning phase, systems may be evaluated with different, potentially competing objectives and institutional policies or communication delays may prevent closely-coupled design. During the operations phase, competition in both supplier and customer markets creates competitive relationships between members and effective and affordable (from a power and processing perspective) cryptography will be crucial to serve customers with strict privacy requirements. Second, established design processes such as systems engineering cannot be directly applied to FSS as there is no central authority. Design activities at the SoS level are limited to establishing interfaces between constituent systems and creating mechanisms to achieve desired collaborative behaviors [4].

To better understand barriers to collaboration and progress towards interface definition and mechanism design, our approach draws from interactive simulation (gaming) methods applied in other domains to combine explicit technical models with implicit social models in participating human "players." This approach was first developed as military wargaming whereby real stakeholders wage a simulated conflict in an interactive, experiential simulator. Applied to FSS, stakeholders are partially-competitive and partially-cooperative, seeking to establish a feasible and viable federation for mutual benefit. A simulator communicates technical model implications and the simulation execution supports collaboration and sharing of mental models between participants. The objective of this paper is to establish an initial FSS simulator architecture suitable for multi-stakeholder interactive simulation and implement a prototype in sufficient detail to allow others to contribute and interoperate FSS member simulators.

2. FSS SIMULATION ARCHITECTURE

The simulation architecture should parallel the FSS structure to identify and address integration and collaboration challenges early in the FSS conceptualization period. In other words, whereas the FSS is a SoS of member spacecraft, the FSS simulation should be a SoS of member spacecraft

---
[2]The exchanging function is distinguished from transporting as a transfer of resources across system boundaries defined by independent control.

simulations. Therefore, the simulation architecture must support distributed control of member simulators while enabling information exchanges in a time-synchronized simulation.

The High Level Architecture (HLA) [5] is a software architecture for simulation interoperability meeting these objectives. It was originally created to support military simulations (federation executions) with heterogeneous simulators (federates) developed by different equipment suppliers and controlled by multiple service branches. It includes support for synchronization algorithms required to maintain consistent state across the distributed applications and can operate in multiple time advancing modes including real-time, scaled time, and as-fast-as-possible. See [6] for more information on the history, details, and applications of the HLA.

As the HLA is a general-purpose simulation architecture suitable for any application, it must be tailored to the FSS case. This section specifies the federation interfaces to orbital and surface elements and radio transmitters, receivers, and signals as the primary entities in a FSS simulation. These concepts, illustrated in Figure 1, are formalized in a federation object model (FOM) to specify data structures and a federation agreement to define the expected behavior of each component simulation.

Element Interface

The main entities in a FSS simulation include space-based satellites and ground-based stations. They are represented as distinct objects classes inheriting common properties from an abstract FSSelement superclass. The "FSS" prefix used here distinguishes federation objects and data types from those within federates which implement a localized version adhering to the federation interface. By controlling the federation interface, the federate-specific versions can introduce additional attributes, modify the internal state storage mechanism, compose objects in new ways, and use any implementation language and platform supported by the HLA runtime infrastructure.

The FSSelement abstract class includes properties required of any element. These include a text-based element name, position and velocity vectors in Cartesian coordinates, and the reference frame to allow conversion to and from other frames. For example, surface elements usually use an Earth-fixed reference frame while space-based elements use an Earth inertial frame to simplify orbital propagation computations. Predefined values include EME2000 (J2000) as an Earth-fixed frame, ITRF2008 (with and without tidal effects) as an Earth inertial frame conforming to IERS 2000 conventions, and TEME as an alternative Earth inertial frame for propagation using the SGP4 model.

The FSSorbitalElement class adds Keplerian orbital elements to the base FSSelement class. While redundant with the Cartesian position and velocity in the associated reference frame, the additional attributes ease debugging and simplify interpretation of orbital motion. The selected orbital elements include eccentricity, semimajor axis, inclination, longitude of ascending node, argument of periapsis, and mean anomaly.

The FSSsurfaceElement class adds geodetic position properties to the base FSSelement class. While also redundant with the inherited position, the additional attributes ease debugging and the altitude value provides some robustness to differing globe and/or terrain models. The selected geodetic properties include latitude, longitude, and altitude.

Table 2. **Extended Transmitter and Receiver Attributes**

	Encoded in type	Encoded in state
Transmitter	frequency (GHz)	power (W)
	diameter (m)	
	antenna gain (dBic)	
	passive loss (dB)	
Receiver	diameter (m)	line loss (dB)
	antenna gain (dBic)	system noise (dBk)
		received power (dBm)

Communication Interface

The simulation architecture includes a logical communication model to support the information exchanging functions in a FSS. It is based on the Radio Communication Protocol (RCP) in IEEE Std. 1278 Distributed Interactive Simulation (DIS) [7]. While DIS is a predecessor to HLA, no application-specific models are carried over as the HLA, thus we adopt portions of the RCP to this particular HLA application.

The logical radio model has three interdependent components. A transmitter is associated with a controlling element and is responsible for providing transmission state attributes. The RCP recommends attributes including transmitter type, operational state, source of radio input, antenna location and pattern parameters, center frequency and bandwidth, average power, modulation type and specific parameters, and cryptographic equipment. As an initial definition, the FSSradioTransmitter only includes text-based type and state attributes for static and dynamic properties, respectively.

A receiver is associated with a controlling element and a linked transmitter. The RCP recommends attributes including receiver type, operational state, and average power received. As an initial definition, the FSSradioReceiver also only includes text-based type and state attributes.

A signal is the RCP abstraction of a transaction between one transmitter and all receivers capable of receiving its transmission. Once issued, a signal is initially visible to all receivers. Each receiver then determines if the signal contents can physically be received based on associated properties of the source transmitter and sending element and applies any required changes such as noise or error. The RCP recommends signal parameters such as encoding scheme, sampling rate, and bit-level data. As an initial definition, the FSSradioSignal only specifies the complete signal contents encoded as a text-based string. Unlike other objects which are persistent during a simulation execution, the FSSradioSignal is transient and is represented as an HLA interaction class rather than an object class.

While the transmitter and receiver models only include type and state attributes, additional data can be encoded in the text-based representation. For example, the type string may specify static attributes using a JSON format such as `{frequency:27.5, diameter:0.5}` where frequency is measured in gigahertz and diameter in meters. Similarly, the state string may specify dynamic attributes using a JSON format such as `{power:1.0}` where power is measured in watts and a value of 0.0 is equivalent to a disabled state. Table 2 lists several such attributes which may be considered in each application to support calculation of link budgets.

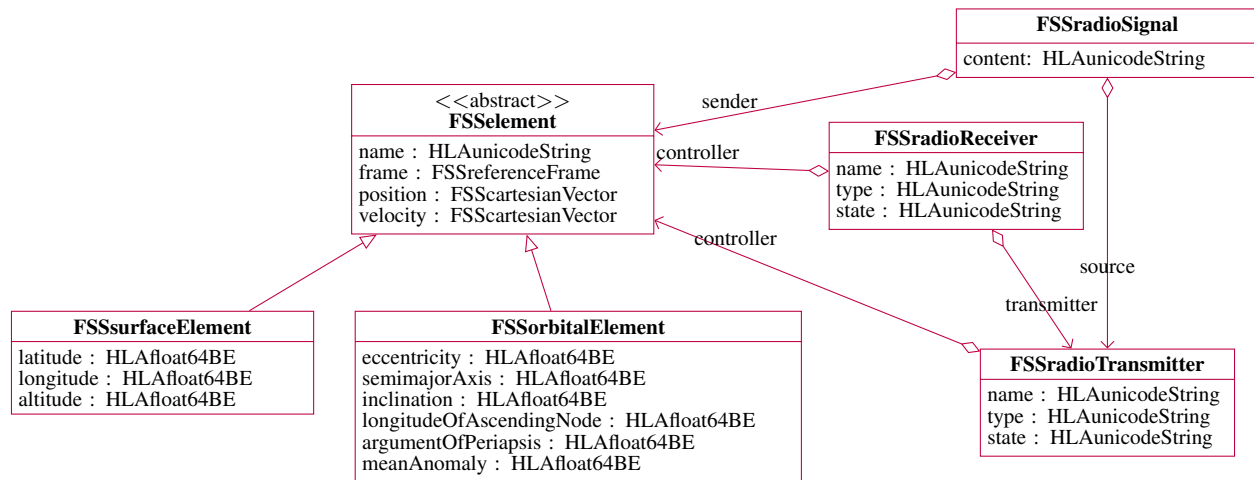Finally, it should be emphasized that the RCP model only

Figure 1 – Object class diagram of the FSS simulation architecture

captures point-to-point messages between transmitters and receivers. Higher level constructs of network protocols such as packet routing would require multiple signals in succession between source-destination pairs. Such an application would also require additional data to be encoded in the contents of the signal, such as that used in terrestrial TCP/IP networks. Implementation of advanced network routing would also require small simulation time steps to capture the behavior on such short time-scales.

Federation Object Model

The element and communication interfaces are composed in a federation object model (FOM) with the following points:

1. FSSelement, FSSorbitalElement, and FSSsurfaceElement are defined as object classes.
  (a) The position and velocity attributes are encoded as fixed array data types with three floating point components.
  (b) The reference frame attribute is encoded as an enumerated data type.
2. FSSradioTransmitter is defined as an object class with a name-based link to the associated controlling element.
3. FSSradioReceiver is an object class with a name-based string for the associated controlling element and transmitter.
4. FSSradioSignal is an interaction class with name-based string for the associated sending element and source transmitter.
5. All text string values use the HLA-default Unicode data type (HLAunicodeString).
6. All floating point values use the HLA-default 64-bit big-endian floating point data type (HLAfloat64BE).
7. Simulation time uses the HLA-default 64-bit floating point data type (HLAfloat64Time).[3]

Detailed FOM tables are provided in Appendix A.

Federation Agreement

While not formally a component of the HLA, the federation agreement follows best practices in distributed simulation in [8] to define behaviors expected of each federate. The FSS federation requires member federates to be responsible for:

1. Maintaining orbital state using propagators,

---

[3]Floating point time is selected for compatibility with the open source Portico HLA library which does not implement all standard features.
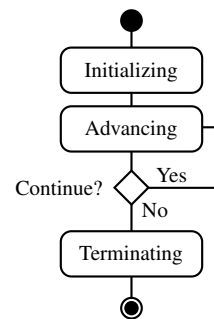

Figure 2 – Activity diagram of a FSS simulation federate

2. Maintaining surface state using globe and terrain models,
3. Determining whether signals are received based on the state of the sending element and source transmitter, and
4. Transforming state between compatible reference frames.

The federates must also interact with the HLA services in a particular procedure to participate in the federation. Figure 2 outlines three activities during a federate life-cycle. The initializing activity sets up a federation execution, the advancing activity sends and receives data between federates, and the terminating activity destroys a federation execution. Each activity is discussed in terms of the specific HLA services used below.

The initializing activity detailed in Figure 3 configures the connection to a federation execution. Orange boxes are RTI-ambassador functions, gray boxes are FederateAmbassador callbacks, and the yellow box interacts with local objects. Federates must connect to the HLA RTI and create and/or join an existing federation execution. Additional options set time constrained behavior (the federate cannot advance beyond simulation time) and time regulating behavior (i.e. no other time constrained federates can advance beyond a specified lookahead period). Initialization also sets publish/subscribe parameters for any desired object and interaction classes. If the federation time is earlier than the federate's initial time, it advances to the initial time. Finally, the federation initializes all local objects to the initial time.
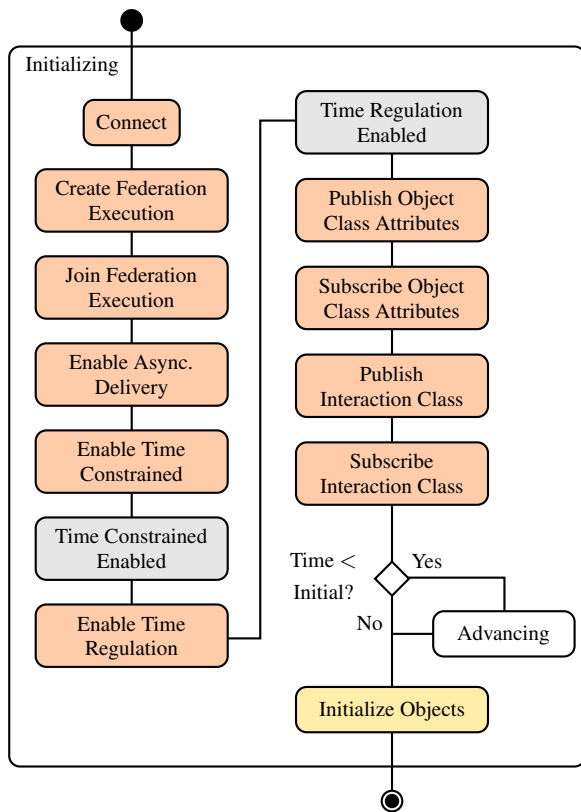
4

Figure 3 – Activity diagram of the initializing process
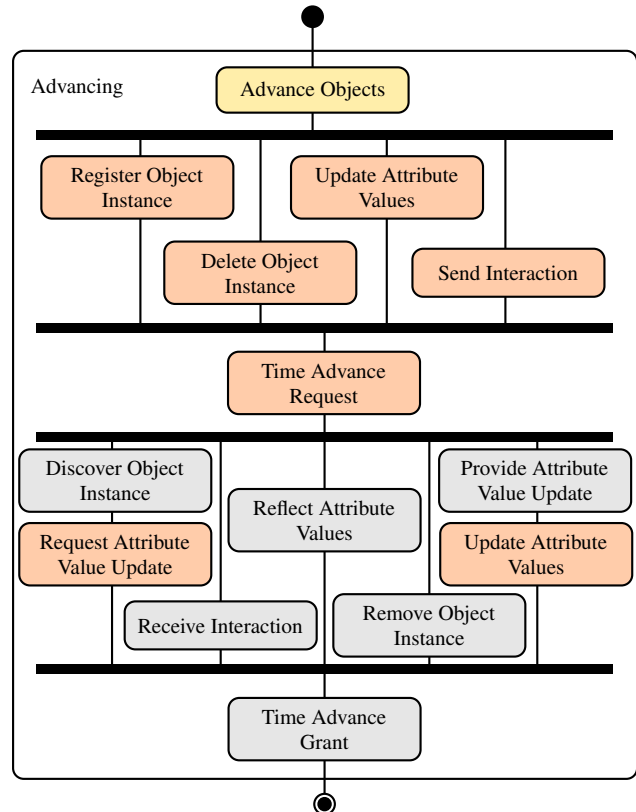

Figure 4 – Activity diagram of the advancing process

The advancing activity detailed in Figure 4 sends and receives updates with the federation while advancing time. First, any local objects are advanced to the next time in preparation to send updates. Next, methods to register new objects, update or delete existing objects, or send interactions take place before a time advance is requested. Once the time advance is requested, callbacks notify of new remote objects (which may require update requests), updates to or removal of existing objects, requests to provide updates of local objects, and new interactions to be received. A time advance is granted once all necessary notifications are received.

Finally, the terminating activity in Figure 5 resets configurations set during initialization. It disables time constrained and regulating behaviors, resigns from the federation execution and requests its destruction if no other federates are still joined. Lastly, the federate disconnects from the HLA RTI.

In addition to the above activities, federates should also agree on an approximate initial time and expected time step durations in simulated and wallclock time. A common initial time allows federates to join a federation in any order and avoid waiting for a prior-joining federate with an earlier initial time to advance to a later time. Setting an initial time may also have implications for gathering ephemeris data for existing satellites. Each time step involves two measurements of time: the duration of simulated time *and* wallclock (real) time elapsed. As all federates are time constrained and regulating, the federation progresses at the rate of the slowest federate. Alignment of simulated time step and expected wallclock duration (e.g. 100 wallclock milliseconds for 1 minute of simulated time) ensures similar performance of each federate.
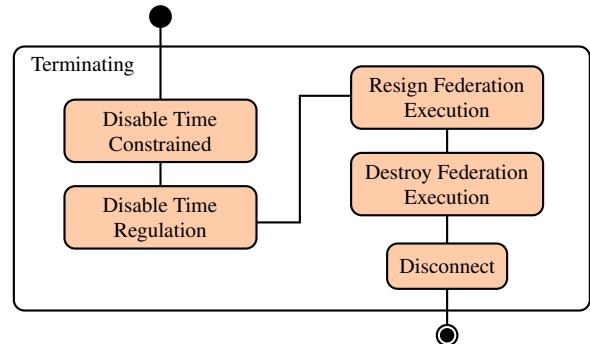

Figure 5 – Activity diagram of the terminating process

3. SAMPLE FSS FEDERATES

This section presents two sample federates implemented in the Java language to demonstrate use of the FSS simulation architecture. The first federate focuses on data visualization and does not control any federation objects. The second federate controls both surface and orbital elements and their associated radio components for application to various FSS use cases.

Both federates use a common structure illustrated in Figure 6 to interact with and simplify HLA interfaces. Here, the HLA interface defines orange boxes, the FSS federation interface defines gray boxes, and yellow boxes are specific to each federate. In particular, the DefaultAmbassador class implements the (HLA) FederateAmbassador interface to send and receive messages from the (HLA) RTIambassador as
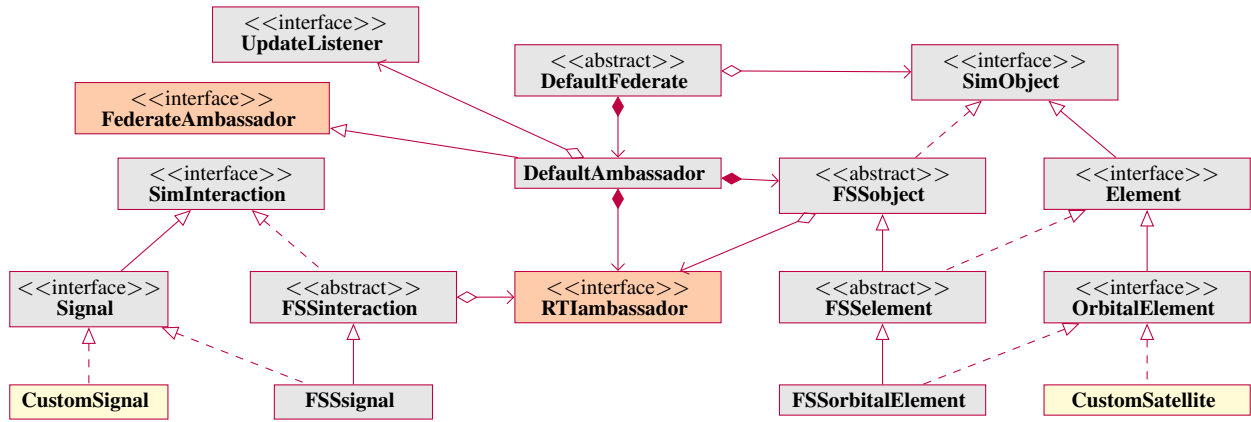
Figure 6 – Simplified object class diagram for a common implementation of both example federates

the logical link to the federation. The DefaultAmbassador composes object classes such as FSSobject, FSSelement, and FSSorbitalElement which represent remotely-controlled objects. These classes use HLA-compatible data types for member variables to simplify updates with the RTIambassador. A similar relationship exists for the FSSinteraction and FSSsignal objects which also use HLA-compatible data types.

Locally-controlled objects within each federate are specified in separate object classes (e.g. CustomSignal and Custom-Satellite) conforming to a common interface specification (e.g. Signal and OrbitalElement) with the remote object classes based on the simulation architecture. The top-level class DefaultFederate aggregates all of the simulation objects to coordinate updates with the DefaultAmbassador. Finally, the DefaultAmbassador sends updates using an observer pattern [12] implemented as an UpdateListener interface

Visualization Federate

The visualization federate uses the NASA World Wind open source project [9] to create a 3D display of the Earth and its immediate vicinity. Although not originally designed for space-based visualization, the World Wind software development kit (SDK) adds rendered objects and modifies the view to simulate an inertial frame. In addition to the graphical output, World Wind also incorporates globe and terrain models which can be used to more accurately display surface elements.

The visualization federate adds a few components to a simple example World Wind application. First, a black surface circle with 50% opacity represents the terminator. Its center is located at the "solar midnight" geodetic position with radius approximated as a quarter the Earth's circumference (the chord length of a 90-degree surface arc). Second, an animation timer rotates the background stars with respect to Earth's rotation. Optionally, the animation can also move the view camera to approximate display from an inertial frame. Finally, the visualization federate registers and displays elements. Spherical ellipsoids mark orbital vehicles with an exaggerated size scale and surface circles optionally highlight regions within a specified field-of-view angle. Cylindrical markers illustrate ground stations.

Figure 7 illustrates a screen capture of the visualization federate showing one orbital element and one surface element. The interface is controlled with mouse drags and key strokes, al-
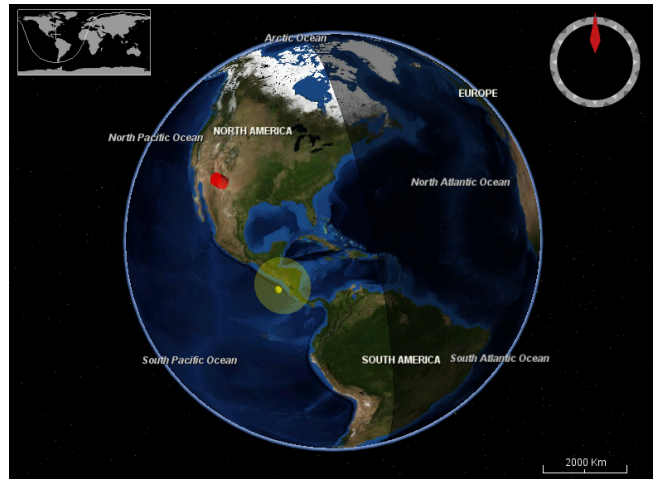

Figure 7 – Screen capture of the World Wind visualization federate

lowing full rotation, zoom, pan, and tilt functions. Additional user interface options (not shown) allow for color and shape customization and federation controls including initializing, advancing, and terminating processes.

The World Wind SDK has a few limitations for space visualization. First, intended for surface locations, markers are only visible if the corresponding geodetic point is also visible, causing spacecraft at high altitudes to disappear when orbiting on the far side. Second, there is an inherent maximum render distance which limits display of interplanetary spacecraft and other celestial bodies. Other projects such as JSatTrak [10] successfully modified the World Wind source code to add a sun object with custom OpenGL shading rather than relying on a surface circle. Future work may also add a 2D ground track visualization to support a complete view of all spacecraft operations.

The Orekit space flight dynamics library [11] manages reference frames and provides celestial body positions in the visualization federate. The orientation of default World Wind model (WW) is related to the Earth-centric fixed (ECF) ITRF2008 frame by a transformation with z-x-z Euler angles
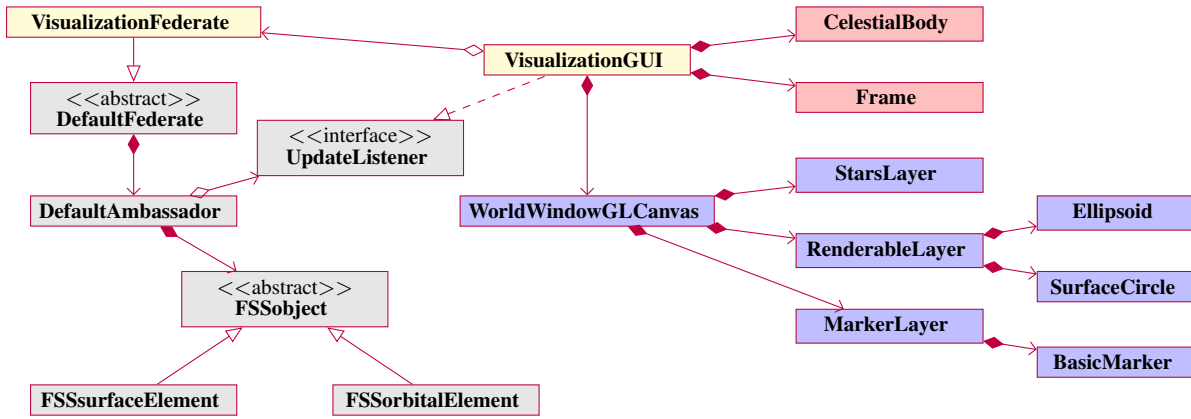
Figure 8 – Simplified object class diagram of the visualization federate

$\alpha = \pi, \beta = \pi/2, \gamma = \pi/2$ such that

$$R_{ECF \to WW} = Z(\alpha)X(\beta)Z(\gamma) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}. \quad (1)$$

All transformation between Earth-centric fixed, World Wind model, and geodetic positions are managed by Orekit classes.

Figure 8 illustrates a simplified object class diagram of the visualization federate. World Wind provides blue boxes, Orekit provides red boxes, the FSS federation interface defines gray boxes, and the remaining yellow boxes are federate-specific implementations. The VisualizationGUI class aggregates the VisualizationFederate and receives updates via the UpdateListener interface. The GUI composes Orekit classes CelestialBody (for sun position tracking) and Frame (for reference frame transformation) and a WorldWindowGLCanvas. Customized layers include a StarsLayer which rotates in the Earth fixed frame, a RenderableLayer to display orbital elements and the terminator, and a MarkerLayer to display surface elements.

FSS Member Federate

The FSS member federate controls space- and surface-based systems participating in the FSS. It uses the Orekit open source space flight dynamics library [11] to calculate geometric properties and perform orbital propagation. Systems compose orbital or surface elements and associated transmitters and receivers as communication subsystems. A more detailed subsystem model illustrated in Figure 9 could add computer and power subsystems for information exchange, transport, processing, and storage as FSS services.

Figure 10 illustrates an object class diagram of the FSS member federate. Orekit provides red boxes, the FSS federation interface specifies gray boxes, and the remaining yellow boxes are federate-specific implementations. Here, the MemberFederate class aggregates all local objects to coordinate updates with the DefaultAmbassador. The OrekitOrbitalElement class relies on several Orekit classes. A Propagator updates its spatial state based on one of several propagation model implementations while an EclipseDetector determines partial or total eclipse. Orekit position transformations calculate slant range to other elements in compatible reference frames. The OrekitSurfaceElement class also relies on the Orekit TopocentricFrame class to calculate azimuth and elevation angles and slant range to other elements in compatible frames.
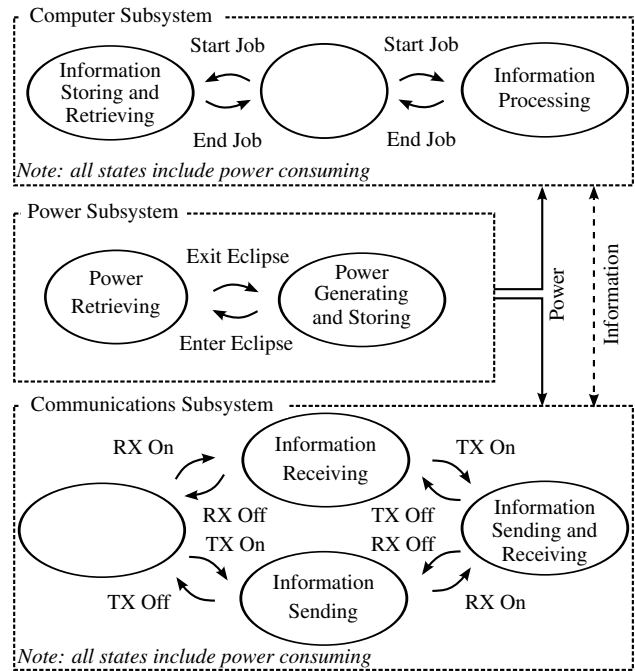

Figure 9 – State diagram of computer, power, and communications subsystems

Owing to differences in reception between satellites and ground stations, both OrbitalReceiver and SurfaceReceiver extend a base DefaultReceiver class but have different detailed implementations. These subclasses can access the slant range and elevation/azimuth calculations in the associated element implementations to override methods to determine if signals can be received from a source element or transmitter. For example, whereas satellite reception is governed by slant range and direct line-of-sight, station reception is governed by slant range and elevation angle.

System classes compose multiple related objects in a single entity. The SurfaceSystem class inherits from OrekitSurfaceElement and composes a SurfaceReceiver and DefaultTransmitter. Similarly, a SpaceSystem class inherits from OrekitOrbitalElement and composes a OrbitalReceiver and DefaultTransmitter. Of particular interest to space systems, this class adds methods to calculate power consumption and
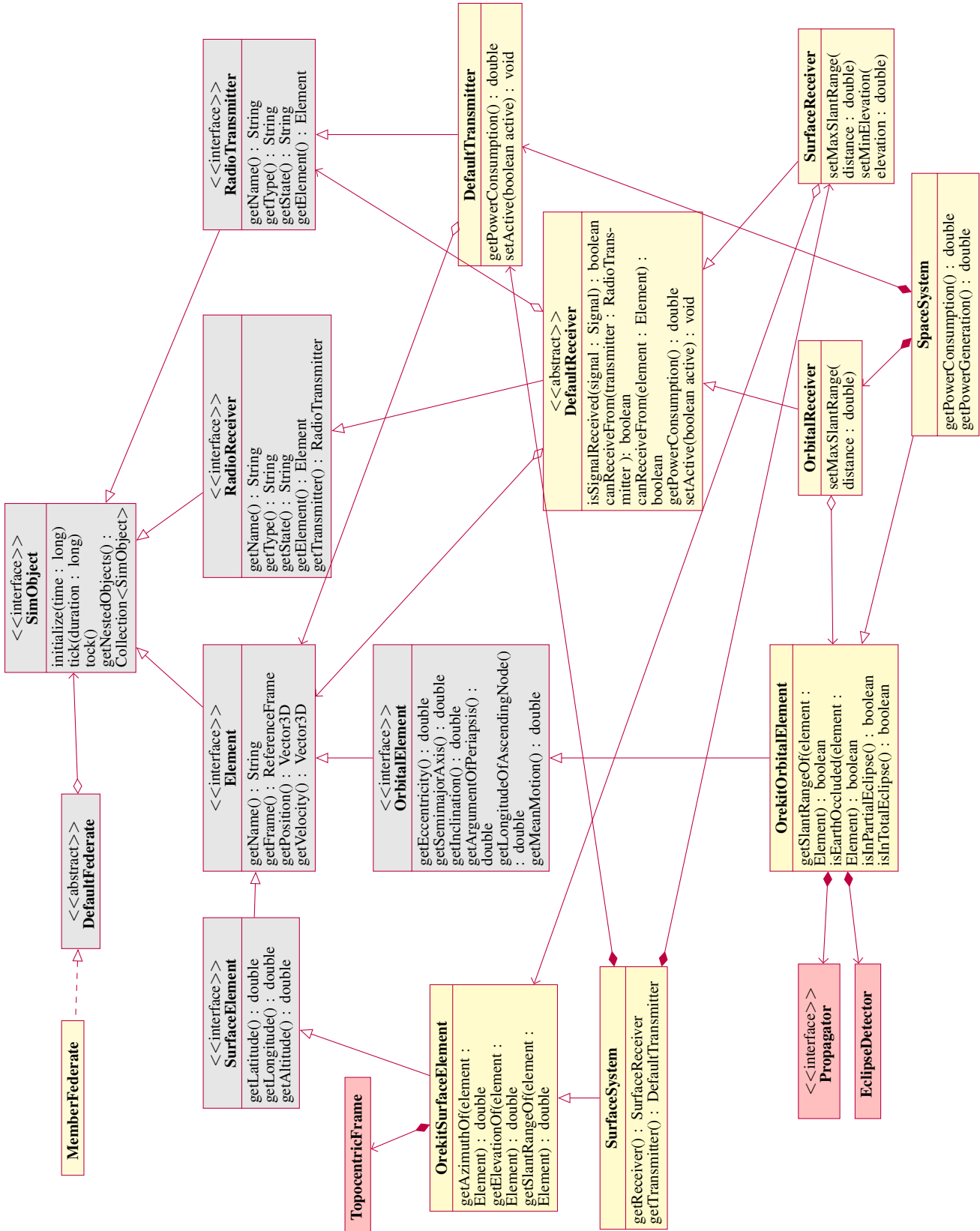
Figure 10 – Partial object class diagram of the FSS member federate

generation based on radio receiver/transmitter state and other subsystem models.

Finally, the FSS member federate also provides opportunities for stakeholder interaction, limited in the current implementation to enabling/disabling the transmitter and receiver state and sending signals. During a simulation, graphical user interface components (not shown in Figure 10) receive user input and communicate actions to the DefaultAmbassador by way of the MemberFederate. Future extensions may allow additional on-orbit operations such as orbital maneuvers and more detail to control information storage and processing.

4. PROTOTYPE FSS SIMULATION

A prototype FSS application applies the simulation architecture and sample federate designs to model a notional case study described in [13]. This concept uses the International Space Station (ISS) as a potential supplier of resources including data storage, on-board processing, and data relay. In particular, large power generation infrastructure aboard ISS—eight solar arrays totaling 84 kilowatts [14]—gives it a sizable advantage over other supplier platforms for power-intensive activities such as processing and transmission.

The objective of the prototype is to validate the applicability of the simulation architecture presented in this paper, demonstrate applications of the sample federates, and provide initial assessment of the ISS-supplier FSS use case based on data link accessibility. It includes four federates: one supplier and two consumers based on the FSS member federate and one optional visualization federate. While this prototype emphasizes the simulation architecture rather than its components, future extensions could add detail. In particular, analysis of link budgets and data rates requires many of the extended transmitter and receiver attributes described in Table 2.

FSS Supplier Federate

The FSS supplier federate models hardware hosted on the ISS capable of exchanging data with customers. FSS services derived from data exchange include information transportation, storage, and processing. The supplier federate defines a SpaceSystem object instance to model on-orbit assets and two SurfaceSystem object instances to model notional ground-based assets under the assumption that official ISS data communication via the Tracking and Data Relay Satellite System (TDRSS) may be limited for third parties such as a FSS supplier.

The SpaceSystem object instance is initialized with an ISS ephemeris from [15] shown in Table 3. It uses an Orekit TLEPropagator propagator to properly leverage the SGP4 propagation model and TEME reference frame for two-line elements.[4] The power subsystem model indicates generation while in sunlight and no generation while in partial or total eclipse. The communication subsystem model measures connectivity with other elements via direct line-of-sight and can receive signals if the slant range to the source transmitter is less than 5123 kilometers, as suggested in [13]. This simplification assumes all transmitters are identical such that signals can be received up to the maximum slant range.

The SurfaceSystem object instances are located at geodetic

coordinates 35.551929° N, 139.647119° E, nominal altitude and 55.698679° N, 37.571994° E, nominal altitude. While neither correspond to active ground stations, they serve as notional locations of third party data links. Both communication subsystem models also consider 5123 kilometers as a maximum slant range and also require at least a 5 degree elevation angle to receive signals.

FSS Consumer Federates

Satellites in a low-Earth or sun-synchronous orbit (SSO) are identified by [13] as favorable for FSS applications due to small slant ranges compared to those in MEO or GEO, a driving requirement for transmitter sizing. High data rate Earth observation satellites in a SSO are targeted as initial FSS customers. This type of satellite has a high data requirement and relatively long revisit time to specific ground stations and the information transportation, storage, and processing services provided by a FSS would be desirable to accommodate new missions. As a placeholder for future systems, two existing satellites are used to approximate orbital characteristics.

Both customers define a SpaceSystem to model the consumer hardware. The first customer uses orbital parameters based on COSMO-SkyMed 1, an Earth observation satellite operated by the Italian Space Agency. The second customer uses orbital parameters based on TerraSAR-X, an Earth observation satellite operated by the German Aerospace Center (DLR) and EADS Astrium. Two-line elements are shown in Table 3 from [15]. Both customers propagate state using the Orekit TLEPropagator and, as they operate in a SSO along the terminator, never enter eclipse. Similar to the supplier federate, the communication subsystem includes a maximum slant range of 5123 kilometers and a direct line-of-sight to receive signals.

Visualization Federate

The visualization federate is unchanged for application to this case study. Figure 11 shows a sample screen capture during simulation execution with the supplier assets including hardware on the ISS and two ground stations and two customers.

Sample Analysis

Analysis of the FSS simulation focuses on the access periods between elements controlled by independent federates. In lieu of a detailed operational analysis of the space-based communication network, only possible access periods are considered based on maximum slant range and line-of-sight conditions. No detailed power, computer, or communications subsystem operation is considered aside from identifying periods of possible power generation for the ISS. The analysis is automated such that there are no user inputs during the execution and no actual signals are issued. The simulation period is initialized at the latest member epoch (i.e. October 14 2013, 4:51 UTC) and runs for 14 days. All access periods are accurate within the simulation time step of one minute. Figure 12 illustrates the main outputs from the simulation including:

1. Possible access periods between the supplier spacecraft (ISS) and customers
2. Power generation periods for the supplier spacecraft (ISS)
3. Possible access periods between the supplier ground stations and spacecraft (ISS)

Table 4 characterizes access periods between the supplier spacecraft and satellites as bi-modal over the simulation time frame. Mode 1 is the short timescale behavior and Mode 2

---

[4]Two-line elements are defined specifically for SGP4 and do not exactly coincide with Keplerian elements. Fortunately, Orekit calculates approximate Keplerian elements as a product of propagation.
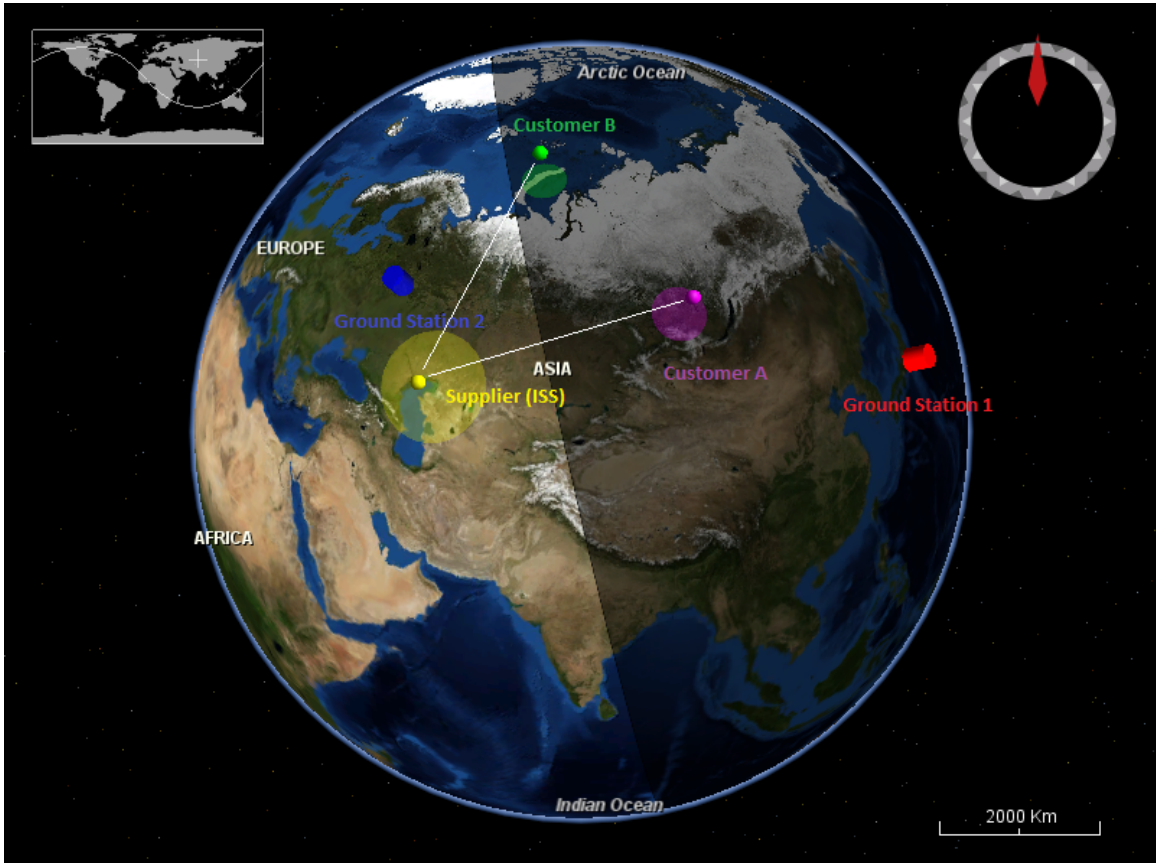
Figure 11 – Screen capture of the visualization federate during the simulation execution at October 20 2013, 12:06 UTC
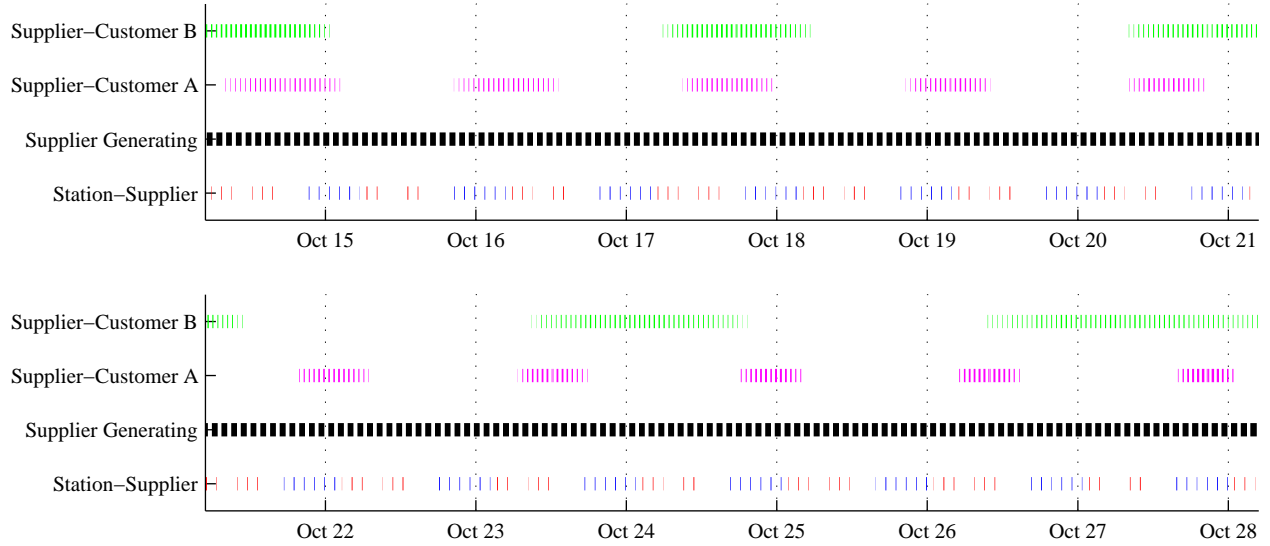

Figure 12 – Access periods greater than one minute in duration in a 14-day period from October 14–28, 2013

Table 3. Prototype FSS Member Two-line Elements

Spacecraft	Supplier	Customer A	Customer B
TLE Entity	ISS (Zarya)	COSMO-SkyMed 1	TerraSAR-X
Epoch (UTC)	2013-10-11 13:08:54.131	2013-10-14 04:51:44.205	2013-10-13 23:15:50.534
Eccentricity (-)	.00011176	.00000313	.00001779
Inclination (°)	51.6510	97.8752	97.4461
Perigee Argument (°)	19.0996	87.3024	93.9122
R.A. Ascending Node (°)	251.7423	110.5203	292.8164
Mean Anomaly (°)	89.0709	272.8359	266.2311

is the long timescale behavior. For example, communication is possible between the supplier spacecraft and Customer A during a span of 12–20 hours, followed by a span of 18–24 hours without access. The mean access window duration is 10.7 minutes with a mean of 32.2 minutes between windows.

These results indicate interactions between a LEO supplier and a SSO customer provide regular communication with maximum wait times of about 30 minutes for about 50% of the orbital period. It also indicates that a single ground station is not likely feasible for a FSS supplier to down-link data, limited to infrequent (4-6 per day) and short-duration (around 5 minutes) access periods. The combination of two spatially-separated ground stations provides superior down-link capabilities with only a few hours each day without regular access. Indeed, direct communication from ground station to a customer produces similar system-level performance without relaying to the supplier spacecraft.

To further investigate the architectural implications of FSS, Figure 13 inspects a 36-hour period between October 20-21, 2013 in detail. Two key features are apparent. First, access periods between the supplier spacecraft and the customers generally coincides with periods of power generation. This positive emergent behavior may allow higher supplier-side power consumption during communication periods. Its effect emerges from pairwise interactions between customers, which never enter eclipse to maintain power generation, and the supplier which has must have a close proximity for communication. Second, the access periods for the two SSO customers tend to coincide with each other. This negative emergent behavior may cause service bottlenecks. Its effect emerges from the composition of interactions between customers sharing similar orbits and the supplier.

The potential for a supply-constrained service introduces interesting operational dynamics in a FSS. If based on a business case for information transportation, storage, and processing, market-based auction mechanisms may be possible to optimize services where FSS customers bid on upcoming opportunities for service. Future analysis may introduce agent-based modeling methods to evaluate such mechanisms in more detail. Implementation within the FSS simulation architecture would introduce new interactions between federates to allow FSS suppliers to identify future service opportunities and solicit bids from customers, each having independent control over the bidding method.

While only an initial prototype demonstration of a FSS simulation, this application case highlights the impacts of the FSS architecture. Namely, even though the two customers do not have direct contact with each other, interactions arise through shared use of a supplier service. Initial results indicate a supplier hosted on the ISS could provide regular access for half of an orbital period for a SSO customer. Finally, a FSS supplier may require multiple ground stations to reduce maximum wait times for space-to-ground signal communication.

5. CONCLUSION

This paper presents initial work towards a multi-stakeholder interactive simulation to study FSS as a SoS. The distributed simulation architecture, based on the HLA standard, defines orbital and surface elements and radio transmitters, receivers, and signals to perform information exchanging functions. Sample federate implementations in the Java language use the existing World Wind and Orekit open source libraries to model space and surface assets and provide 3D visualization. Finally, a prototype application validates the simulation architecture in a conceptual FSS using the ISS as a service provider. Analysis results find emergent effects between FSS members based on common features of customer orbits.

While a similar FSS analysis could have been conducted in a much simpler centralized simulation, the federated simulation approach mirrors the structure of FSS to provide distributed authority over the simulation components. This has several implications. First, independent control of federate simulators allows decoupled development to a well-defined interface (namely, that presented in this paper). Second, the information specified in the federation interface is the only connection between a federate and the federation. This allows all implementation details of a spacecraft model to be kept private for security, proprietary, or other reasons. For example, the procedure to bid for auction-based service should be kept private from potentially competing customers. Finally, any level of detail can be added to federate models such as power and communications subsystems, or even integration of real or prototype hardware.

Future work may extend this paper in several ways. First, federate simulators for both supplier and customers should include more detailed models of power, computer, and communications subsystems. For example, a power budget must consider the FSS transmitter and receiver switching states and the communications subsystem should accurately model transmission power and other parameters including the impact of encryption on performance. In the limit, prototype hardware may be integrated in a real-time simulation loop, as is done for other HLA applications. Second, additional FSS functionality should be added to the base signal interaction. A message protocol may be established to request and provide FSS services such as computation or data relay, such as in an auction-based mechanism. Finally, future work seeks to apply a version of this prototype in interactive sessions with real or role play stakeholders. These design experiments would study the social dimension of collaborative processes for planning FSS with competing local objectives among participants.

Table 4. **Bi-modal Access Period Characterization**

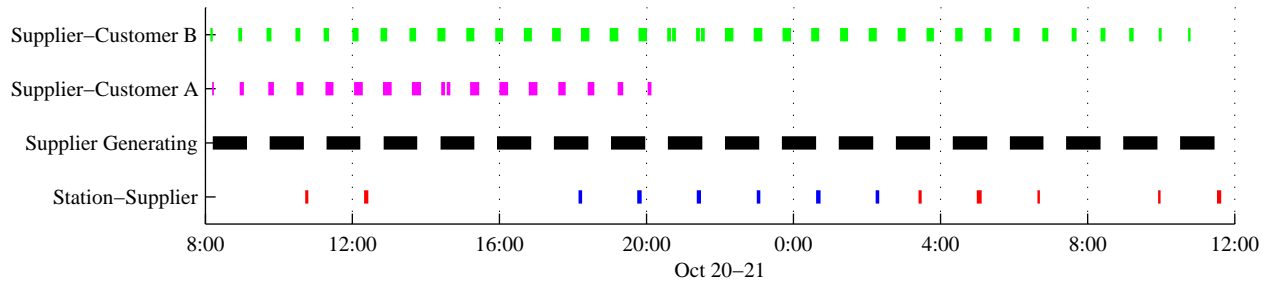|	Mode 1 Times (min)		Mode 2 Times (hr)	
FSS Pair	Mean Access	Mean Revisit	Approx. Access	Approx. Revisit
Supplier–Customer A	10.7	32.2	12–20	18–24
Supplier–Customer B	10.6	33.9	24–36	36–48
Station 1–Supplier	5.3	88.4	6–8	14–16
Station 2–Supplier	6.5	88.5	6–8	14–16

Figure 13 – **Access periods greater than 1 minute in duration in a 36-hour period between October 20–21, 2013**

APPENDICES

A. FEDERATION OBJECT MODEL

The following tables provide the federation object model (FOM) required for the HLA. Table 5 defines the object classes including Element (and its subclasses SurfaceElement and OrbitalElement), RadioTransmitter, and RadioReceiver. Table 6 defines the RadioSignal interaction class. Table 7 defines the ReferenceFrame enumerated data type and Table 8 defines the CartesianVector data type. Finally, Table 9 defines the time-related data types.

B. EXTENDING THE FSS FEDERATES

The sample FSS federates were developed to be easily-extended to future applications. They are implemented in Java, a platform-independent, mature language with extensive third party libraries. Java is also supported by most HLA implementations including the open source Portico RTI [16] used by the authors. This appendix briefly discusses how to extend the DefaultFederate to use custom simulation objects for the FSS member federate as illustrated in Figure 10.

SimObject is the primary interface to all simulation objects and requires four functions to be completed. The initialize function instructs a simulation object to start at a particular time. The tick function instructs a simulation object to compute, but not yet save, its new state after a specified duration. The tock function saves the new state. The tick-tock pattern allows simulation objects to update without order dependencies. Finally, the getNestedObjects function returns a collection of any contained objects to be included.

The example in Box 1 describes a custom TrivialClock object which only maintains a local time state. The initialize, tick, and tock functions update its state variable as the simulation progresses. Though only a simple example, additional state variables, methods, or nested objects can be added for new functionality. Implementing a custom FSS element requires extending the SurfaceElement or OrbitalElement interfaces which have additional functions to be completed (i.e. those

in Figure 10 such as getName, getPosition, getLatitude, and getEccentricity). This ensures the federate can communicate required information to remote HLA-compatible objects (e.g. FSSsurfaceElement and FSSorbitalElement).

Box 1. Trivial Clock Simulation Object

```java
import java.util.Collection;
import java.util.HashSet;

import edu.mit.fss.SimObject;

class TrivialClock implements SimObject {
   // state var. to store current time (ms)
   long currentTime;

   // temp. var. to store next time (ms)
   long nextTime;

   @Override
   public Collection<SimObject>
      getNestedObjects() {
      // there are no nested objects
      return new HashSet<SimObject>();
   }

   @Override
   public initialize(long time) {
      // initialize state at time (ms)
      currentTime = time;
   }

   @Override
   public tick(long duration) {
      // compute state after duration (ms)
      nextTime = currentTime + duration;
   }

   @Override
   public tock() {
      // save new state
      currentTime = nextTime;
   }
}
```

12

Table 5. FOM Object Class Table

Object	Attribute	Data Type	Update Type	Order	P/S	Semantics
FSSelement	Name	HLAunicodeString	Receive	Static	PS	Unique element name.
	Frame	FSSreferenceFrame	Timestamp	Conditional	PS	Reference frame for coordinates.
	Position	FSScartesianVector	Timestamp	Conditional	PS	Position in meters.
	Velocity	FSScartesianVector	Timestamp	Conditional	PS	Velocity in meters per second.
FSSelement.	Latitude	HLAfloat64BE	Timestamp	Conditional	PS	Latitude in degrees North.
FSSsurfaceElement	Longitude	HLAfloat64BE	Timestamp	Conditional	PS	Longitude in degrees East.
	Altitude	HLAfloat64BE	Timestamp	Conditional	PS	Altitude in meters above sea level.
FSSelement.	Eccentricity	HLAfloat64BE	Timestamp	Conditional	PS	Orbital eccentricity.
FSSorbitalElement	SemimajorAxis	HLAfloat64BE	Timestamp	Conditional	PS	Orbital semimajor axis in meters.
	Inclination	HLAfloat64BE	Timestamp	Conditional	PS	Orbital inclination in degrees.
	LongOfAscendNode	HLAfloat64BE	Timestamp	Conditional	PS	Longitude of ascending node in degrees.
	ArgumentOfPeriapsis	HLAfloat64BE	Timestamp	Conditional	PS	Argument of periapsis in degrees.
	MeanAnomaly	HLAfloat64BE	Timestamp	Conditional	PS	Orbital mean anomaly in degrees.
FSSradioTransmitter	Name	HLAunicodeString	Receive	Static	PS	Unique transmitter name.
	ElementName	HLAunicodeString	Timestamp	Conditional	PS	Name of controlling element.
	Type	HLAunicodeString	Receive	Static	PS	Type of transmitter.
	State	HLAunicodeString	Timestamp	Conditional	PS	State of transmitter.
FSSradioReceiver	Name	HLAunicodeString	Receive	Static	PS	Unique receiver name.
	ElementName	HLAunicodeString	Timestamp	Conditional	PS	Name of controlling element.
	TransmitterName	HLAunicodeString	Timestamp	Conditional	PS	Name of associated receiver.
	Type	HLAunicodeString	Receive	Static	PS	Type of receiver.
	State	HLAunicodeString	Timestamp	Conditional	PS	State of receiver.

Table 6. FOM Interaction Class Table

Object	Parameter	Data Type	Order	Semantics
FSSradioSignal	TransmitterName	HLAunicodeString	Timestamp	Name of source transmitter.
	ElementName	HLAunicodeString	Timestamp	Name of sending element.
	Content	HLAunicodeString	Timestamp	Content of message.

Table 7. FOM Enumerated Data Type Table

Name	Representation	Enumerator	Values	Semantics
FSSreferenceFrame	HLAinteger32BE	Unknown	0	Unknown frame.
		EME2000	1	EME2000 (J2000) Earth fixed frame.
		ITRF2008	2	ITRF 2008 Earth inertial frame using IERS 2010 conventions, no tidal effects.
		ITRF2008_TE	3	ITRF 2008 Earth inertial frame using IERS 2010 conventions with tidal effects.
		TEME	4	TEME Earth inertial frame for two-line element propagation using SGP4 model.

Table 8. FOM Array Data Type Table

Name	Element Type	Cardinality	Encoding	Semantics
FSScartesianVector	HLAfloat64BE	3	HLAfixedArray	Cartesian vector with x-, y-, and z-components.

Table 9. FOM Time Representation Table

Category	Data Type	Semantics
Timestamp	HLAfloat64Time	Absolute time measured in milliseconds since January 1, 1970 0:00:00.0 using the UTC time scale.
Lookahead	HLAfloat64Time	Time duration measured in milliseconds.

Regardless of how detailed custom objects are, they are all added to a federate in the same way to initialize data transfer with other federation members. The example in Box 2 shows how a new CustomFederate (trivially extending the base DefaultFederate class) can be used in a main method to add a new object, connect to a federation, initialize, and run a simulation execution. This example omits configuration options to specify federation connection parameters (e.g. federation name, federate name and type) and has no ending condition or graphical user interface. Any objects to be notified of remote object state changes or interactions can implement the UpdateListener interface and be added as a listener to the federate.

Box 2. Custom Federate with Main Method

```java
import edu.mit.fss.DefaultFederate;

class CustomFederate extends DefaultFederate
    {
  public static void main(String[] args) {
      // the program starts here

      // create a new federate object
      CustomFederate fed = new
          CustomFederate();

      // add a new custom object
      fed.addObject(new TrivialClock());

      // connect to the federation
      fed.connect();

      // do initializing process and
      // initialize SimObject objects
      fed.initialize();

      // tick and tock SimObject objects
      // and do advancing process
      fed.run();
  }
}
```

The examples presented in this paper were implemented using open source and freely-distributable software libraries including Orekit [11], World Wind [9], Apache Math Commons [17], JFreeChart [18], and Portico [16] with the intent of creating an open source toolkit for FSS simulation. Please contact the first author for more information about obtaining the source code and supplemental materials.

ACKNOWLEDGMENTS

REFERENCES

[1] A. Golkar, "Federated satellite systems: an innovation in space systems design," in *9th IAA Symposium on Small Satellites for Earth Observation*. Berlin, Germany: International Academy of Astronautics, April 2013.

[2] O. L. de Weck, D. Roos, and C. Magee, *Engineering Systems: Meeting Human Needs in a Complex Technological World*. MIT Press, January 2012.

[3] N. P. Suh, "A theory of complexity, periodicity and the design axioms," *Research in Engineering Design*, vol. 11, pp. 116–131, 1999.

[4] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[5] IEEE, "IEEE standard for modeling and simulation (M&S) high level architecture (HLA) – framework and rules," 2010, IEEE Std. 1516-2010.

[6] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons, 2000.

[7] IEEE, "IEEE standard for distributed interactive simulation – application protocols," 1998, IEEE Std. 1278.1a-1998.

[8] ——, "IEEE recommended practice for distributed simulation engineering and execution process (DSEEP)," 2010, IEEE Std. 1730-2010.

[9] NASA, "World Wind Java SDK," 2013, accessed 16-Oct 2013. [Online]. Available: http://worldwind.arc.nasa.gov/

[10] S. Gano, "JSatTrak," 2013, accessed 16-October 2013. [Online]. Available: http://www.gano.name/shawn/JSatTrak/

[11] CS Systèmes d'Information, "Orekit: a free low-level space dynamics library," 2013, accessed 16-Oct 2013. [Online]. Available: https://www.orekit.org/

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriebted Software*. Addison-Wesley, 1994.

[13] A. Golkar, "Federated satellite systems: A case study on sustainability enhancement of space exploration systems architectures," in *64th International Astronautical Congress*, no. IAC-13-D3.4. Beijing, China: International Astronautical Federation, September 2013.

[14] NASA, "International Space Station: Facts and figures," 2013, accessed 18-Oct 2013. [Online]. Available: http://www.nasa.gov/mission_pages/station/main/onthestation/facts_and_figures.html

[15] T. S. Kelso, "NORAD two-line element sets: Current data," 2013, accessed 19-Oct 2013. [Online]. Available: http://www.celestrak.com/NORAD/elements/

[16] The Portico Project, 2013, accessed 18-Sept 2013. [Online]. Available: http://www.porticoproject.org/

[17] Apache Software Foundation, "Commons Math 3.2," 2013, accessed 16-Oct 2013. [Online]. Available: http://commons.apache.org/proper/commons-math/

[18] Object Refinery Limited, "JFreeChart," 2013, accessed 16-Oct 2013. [Online]. Available: http://www.jfree.org/jfreechart/

BIOGRAPHY

Paul T. Grogan received his B.S. degree in Engineering Mechanics and Astronautics from the University of Wisconsin – Madison in 2008 and his S.M. degree in Aeronautics and Astronautics from the Massachusetts Institute of Technology in 2010. He is currently a graduate research assistant in the Massachusetts Institute of Technology Engineering Systems Division. His research applies interoperable simulation technologies for collaborative design of complex socio-technical systems.

Alessandro Golkar received his Laurea degree in Aerospace Engineering in 2006 and his Laurea Specialistica degree in Astronautics Engineering in 2008 from University of Rome "La Sapienza" and a Ph.D. in Aeronautics and Astronautics from the Massachusetts Institute of Technology in 2012. He is currently an Assistant Professor at the Skolkovo Institute of Science and Technology. His research interests lie in the areas of systems architecture, project management, systems engineering, and spacecraft design analysis and optimization.

Seiko Shirasaka received his M.S. degree in Astronautics from the University of Tokyo in 1994 and a Ph.D. in Systems Engineering from Keio University in 2012. He previously worked as a systems engineer at the Mitsubishi Electric Corporation for 15 years. He is currently an Associate Professor at the Keio University Graduate School of System Design and Management. His research interests include space systems engineering, methodology of system development, and system safety.

Olivier L. de Weck received his Dipl. Ing. degree in Industrial Engineering in 1993 from ETH Zurich and his S.M. degree in Aeronautics and Astronautics and a Ph.D. in Aerospace Systems from the Massachusetts Institute of Technology in 1999 and 2001. He is currently a Professor of Aeronautics and Astronautics and Engineering Systems at the Massachusetts Institute of Technology. His research focuses on the strategic design of complex systems to deliberately account for future uncertainty and context to maximize lifecycle value.

15