FSS Simulation Toolkit v0.1
User's Guide

Paul T. Grogan*

Revised April 3, 2014

1 Introduction

This user's guide covers the FSS Simulation Toolkit (hereafter "the toolkit") which was developed during academic year 2013-4 as a part of the Federated Satellite Systems (FSS) joint project between the Massachusetts Institute of Technology and Skolkovo Institute of Technology (P.I. Prof. de Weck and Co-P.I. Prof. Golkar). This document extends high-level discussion of the toolkit provided in the initial conference paper:

> P. T. Grogan, A. Golkar, S. Shirasaka, and O. L. de Weck (2014). "Multi-stakeholder Interative Simulation for Federated Satellite Systems". In: *2014 IEEE Aerospace Conference*. Paper #2163. Big Sky, Montana

to include installation and configuration notes and extended discussion of examples and tutorials. The intent of the toolkit is to eventually release it as an open source library under a permissive license at which time this document should be transformed to a community-accessible format such as a wiki.

As discussed in the above conference paper, a FSS is a system-of-systems (SoS) of constituent spacecraft exhibiting resource exchanging behaviors. Each FSS member voluntarily joins out of mutual benefit and no centralized authority can exert control the total SoS. An isomorphic simulation architecture, i.e. one which provides decentralized control over member simulation models, is desired to aid design activities. For example, it may be used to formulate mechanisms for service provision under partial information, to identify integration requirements between FSS members, or simply to distribute conceptual spacecraft designs among various teams.

This toolkit is based on an existing standard for interoperable simulation called the High Level Architecture (HLA, IEEE Standard 1516-2010). The HLA defines the interface between a member simulation model (a "federate") and rest of the SoS simulation (a "federation"). It relies on a runtime implementation (RTI) to manage key processes such as federation, data, object, and time management. As a mature standard, there are several RTI implementations available, both commercially licensed and open source with permissive licensing.

The HLA is an interoperability standard which minimizes assumptions of the detailed implementation of federates. RTI implementations, however, generally only support mature high-level languages such as Java, C, and C++. Based on the development experience of this author and on the wide availability of open source libraries with permissive licensing, Java was selected as the toolkit's implementation language. Other languages could be supported under a FSS federation; however they must be supported by a selected RTI and ports of this toolkit would be necessary to conform to the expected federation object model (FOM) and federation agreement discussed in Grogan et al. (2014).

*ptgrogan@mit.edu

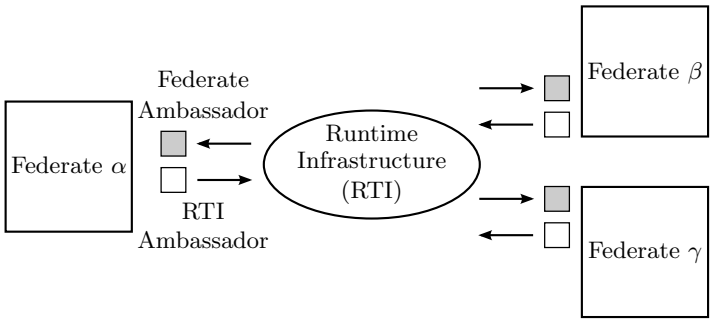

Figure 1: Structure of an HLA federation. Federates receive information via a federate amabssador and send information via an RTI ambassador.

Figure 1 illustrates the overall structure of a HLA federation. The independent federates (α, β, and γ here) communicate with each other according to standard services via the RTI. Data structures within messages are based on the common form specified in the FOM. The particular method of communication is not specified by the HLA standard. Instead, each federate uses a federate ambassador to receive information from the rest of the federation and accesses an RTI ambassador to send information to the federation.

2 Toolkit Installation and Configuration

This section describes how to download and configure the toolkit to run the example federates.

2.1 Installation

As of this time there is no installer script for the toolkit. Instead, it is used within an integrated development environment (IDE) to help manage dependent libraries. There are several excellent Java IDEs which could alternatively be used to develop applications using the toolkit, however this author is most familiar with Eclipse.

1. Download and install the Eclipse IDE:
   (a) Download the Eclipse IDE for Java:
   `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/`
   (b) Extract the .zip file to a desired file location (there is no installer).
   (c) Run the `eclipse.exe` executable and choose a local workspace location.
2. Install the Subclipse extension for integrated SVN access:
   (a) Select "Help > Eclipse Marketplace..." from the Eclipse menu.
   (b) Search for "Subclipse" and click "Install" for a match (version Subclipse 1.10.3 as of writing).
   (c) Select all features, click "Confirm," and accept the license agreement to install.
   (d) Restart Eclipse to complete installation.
3. Checkout the FSS toolkit:
   (a) Select "File > New > Other" from the Eclipse menu.

(b) Expand the "SVN" folder, select "Checkout Projects from SVN" and click "Next" to continue.

(c) Select "Create a new repository location" and click "Next" to continue.

(d) Enter the URL `svn://ptgrogan.scripts.mit.edu/fss/` and click "Next" to continue.

(e) Select the `trunk` directory and click "Finish" to continue.

(f) Expand the "Java" folder, select "Java Project" and click "Next" to continue.

(g) Give the project a name, change other options (if desired, note: Java SE 7 is the minimum version required), and click "Finish" to download the source files.

4. Configure the FSS toolkit in Eclipse:

(a) Select "Project > Properties" from the Eclipse menu, select the "Java Build Path" item from the left-hand menu, and select the "Libraries" tab on the right.

(b) Click the "Add JARs..." button and select the following:

i. `lib/commons-math3-3.2/commons-math3-3.2.jar`
ii. `lib/jfreechart-1.0.17/jcommon1.0.21.jar`
iii. `lib/jfreechart-1.0.17/jfreechart-1.0.17.jar`
iv. `lib/log4j-1.2.17/log4j-1.2.17.jar`
v. `lib/orekit-6.0/orekit-6.0.jar`
vi. `lib/portico-2.0.0/portico.jar`
vii. `lib/worldwind-2.0-857.1737/worldwind.jar`

(c) Click "OK" to add the JARs to the project library.

(d) **Optionally** expand JAR files, select the Javadoc location item, click "Edit...," and select the appropriate Javadoc file to provide documentation within the Eclipse IDE:

i. Commons Math 3: Javadoc in archive, Workspace file:
`lib/commons-math3-3.2/commons-math3-3.2-javadoc.jar`
ii. JFreeChart: Javadoc in archive, Workspace file:
`lib/jfreechart-1.0.17/jfreechart-1.0.17-javadocs.zip`
iii. Log4J: Javadoc in archive, Workspace file:
`lib/log4j-1.2.17/log4j-1.2.17-apidocs.zip`
iv. Orekit: Javadoc in archive, Workspace file:
`lib/orekit-6.0/orekit-6.0-javadoc.jar`
v. World Wind: Javadoc in archive, Workspace file:
`lib/worldwind-2.0-857.1737/worldwind-docs-daily-857.1737.zip`

(e) Click "OK" to close the project properties. The project should re-build automatically and all errors should disappear.

5. Test the example federates:

(a) Select the CosmosSkyMED-1 example federate in the file tree `edu.mit.fss.examples/CosmoSkyMed1.java` and click the green "Run" button in Eclipse.

(b) Click the plug icon, then click the play icon in the GUI to initialize the federate and start advancing time.

(c) Select the Visualization example federate in the file tree `edu.mit.fss.examples/Visualization.java` and click the green "Run" button in Eclipse.

(d) Click the plug icon in the visualization federate GUI; it should halt the federation execution. (Note: if the federates were run in the reverse order, CosmosSkyMED-1 would wait for the time to reach October 2013 before halting the federation.)

(e) Click the play icon in the visualization federate GUI; it should allow time to advance in both federates.

2.2 Network Configuration

Any number of federates can be run on a single computer through the default loop-back interface, but to enable distributed simulation of federates across multiple computers each participant must be connected to a network and configured. The UDP multi-cast protocol used by the Portico RTI may be blocked by shared network infrastructure. Due to these limitations, a local area network (LAN) with a private router may be required to host a distributed simulation.

1. Connect each participant to the LAN (wired or wireless).
2. If there are multiple network interfaces, select the correct bind address for Portico by configuring the `portico.jgroups.udp.bindAddress` property in the `RTI.rid` file (uncomment and add the IP address on the private LAN).
3. Depending on security settings, software firewalls (i.e. Windows Firewall) may need to be disabled or have new rules added to allow UDP traffic.

3 Toolkit Contents

The toolkit is presently stored in a Subversion repository hosted by the M.I.T. Scripts service. Its files and history is publicly-accessible at the URL:

`svn://ptgrogan.scripts.mit.edu/fss/`

This repository is set to anonymous read access and authenticated write access. While anyone can download the files, only authorized developers can make changes. Future development may migrate to an open source repository service to more easily allow community contributions.

The repository contains the following directory structure:

```
svn://ptgrogan.scripts.mit.edu/fss/
├── branches/
├── tags/
└── trunk/
    ├── src/
    │   ├── edu.mit.fss
    │   ├── edu.mit.fss.event
    │   ├── edu.mit.fss.examples
    │   ├── edu.mit.fss.hla
    │   ├── edu.mit.fss.tutorial
    │   └── resources
    ├── doc/
    ├── lib/
    ├── lib-external/
    └── license.txt
```

The root `trunk/` directory contains several configuration and data files. The `src/` directory contains the source code within the `edu.mit.fss` package structure and icons in the `resources` package. The `doc/` directory contains documentation including this user guide. The `lib/` and `lib-external/` directories contain redistributable third-party libraries. Finally, the `license.txt` file contains the standard 2-clause BSD license under which the toolkit is released.

3.1 Core Toolkit Interfaces

The core toolkit interfaces provide the top-level structure and require methods for Java object classes participating in a FSS simulation. The following files define the core interfaces to the FSS simulation toolkit:

```
trunk/
└── src/
    └── edu.mit.fss
        ├── Element.java ..... Interface to an element with Cartesian coordinates
        ├── OrbitalElement.java ..... Interface to an element with Keplerian orbital elements
        ├── RadioReceiver.java ..... Interface to a RCP receiver abstraction
        ├── RadioTransmitter.java ..... Interface to a RCP transmitter abstraction
        ├── ReferenceFrame.java ..... Enumeration of existing reference frames
        ├── Signal.java ..... Interface to a RCP signal abstraction
        ├── SimInteraction.java ..... Interface to a simulation interaction
        ├── SimObject.java ..... Interface to a simulation object
        └── SurfaceElement.java ..... Interface to an element with geodetic coordinates
```

Figure 2 illustrates an object class diagram of the core interface classes. A few classes introduce external dependencies. First, the `Vector3D` data type used in the `Element` type is from the Apache Commons Math 3 library. Second, the `Frame` data type used in the `ReferenceFrame` enumeration is from the Orekit flight dynamics library.

3.2 HLA Model Implementations

The HLA model implementations use the core interfaces to define objects compatible with the HLA runtime infrastructure (RTI). For example, a "remote" object in a federation will appear as a HLA model implementation, as will a copy of a "local" object to facilitate information exchange with the federation. The following classes implement the core interfaces using HLA data types:

```
trunk/
└── src/
    └── edu.mit.fss.hla
        ├── FSScartesianVector.java ..... HLA implementation of Vector3D type
        ├── FSSelement.java ..... HLA implementation of Element interface
        ├── FSSorbitalElement.java ..... HLA implementation of OrbitalElement interface
        ├── FSSradioReceiver.java ..... HLA implementation of RadioReceiver interface
        ├── FSSradioTransmitter.java .... HLA implementation of RadioTransmitter interface
        ├── FSSreferenceFrame.java ..... HLA implementation of ReferenceFrame type
        ├── FSSsignal.java ..... HLA implementation of Signal interface
        ├── FSSsurfaceElement.java ..... HLA implementation of SurfaceElement interface
        ├── HLAinteraction.java ..... HLA helper class for managing interactions
        └── HLAobject.java ..... HLA helper class for managing objects
```

The helper class implementations `HLAinteraction` and `HLAobject` work with the federate ambassador to send and receive updates automatically from the federation. Figure 3 illustrates a simplified object class diagram of the HLA model implementation classes.

3.3 Toolkit Execution

The toolkit execution classes help run a federated simulation. They include both the local simulator (within a `Federate` object), an interface to the federation (via the `FSSambassador` class), and events and listeners to pass information. The following classes provide execution capabilities for the toolkit:

```
trunk/
└── src/
    ├── edu.mit.fss
    │   ├── DefaultFederate.java ..... Default implementation of Federate
    │   └── Federate.java ..... Interface to a federate application
    └── edu.mit.fss.event
```

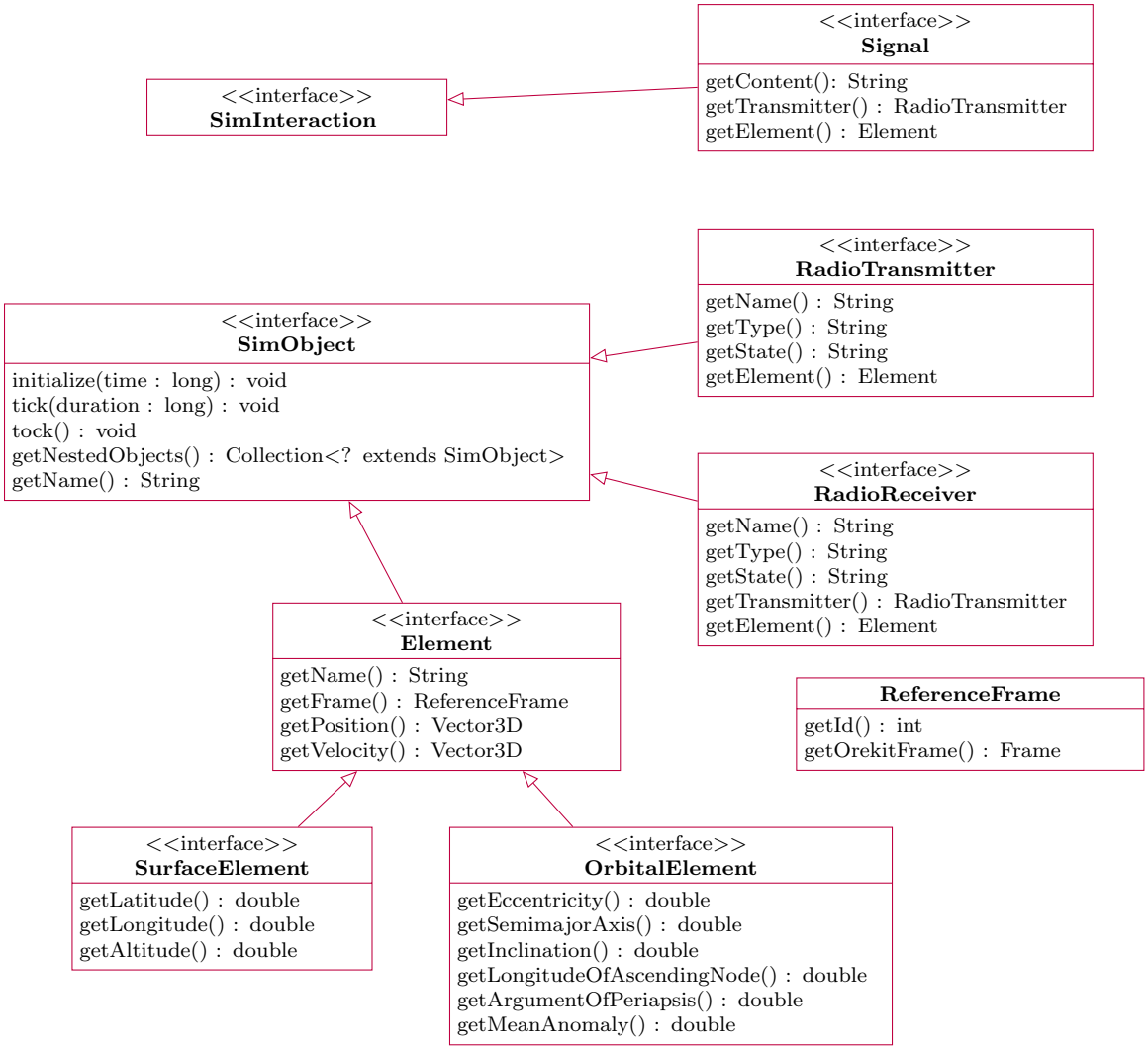

Figure 2: Object class diagram of the FSS simulation toolkit core interfaces

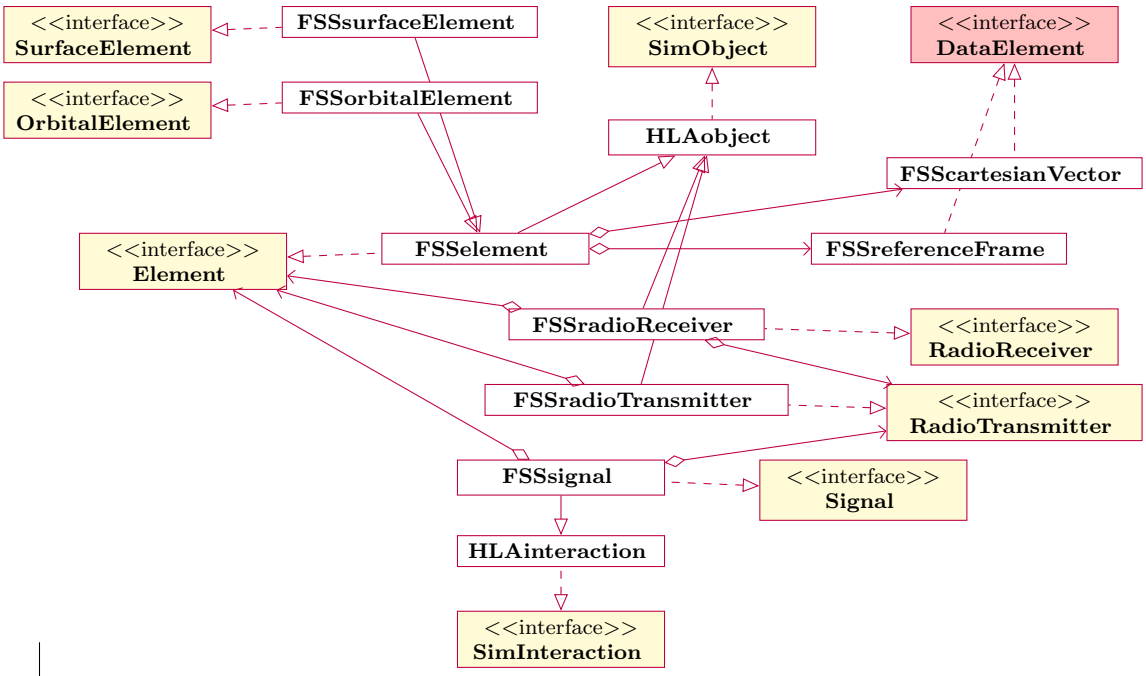

Figure 3: Object class diagram of the FSS simulation toolkit HLA model implementations

```
    ConnectionEvent.java ..... Notifies a connection was changed
    ConnectionListener.java ..... Observer of connection events
    ExecutionControlEvent.java ..... Notifies an execution control action occurred
    ExecutionControlListener.java ..... Observer of execution control events
    ObjectChangeEvent.java ..... Notifies an object or interaction changed
    ObjectChangeListener.java ..... Observer of object change events
    SimulationTimeEvent.java ..... Notifies the simulation time changed
    SimulationTimeListener.java ..... Observer of simulation time events
  edu.mit.fss.hla
    FederationConnection.java ..... Stores state of connection to HLA federation
    FSSambassador.java ..... Interface to a FSS federate ambassador
    DefaultAmbassador.java ..... Default implementation of FSSambassador
connection.data ..... Saved HLA federation connection settings
fss.xml ..... Federation object model (FOM) definition
```

In particular, the `Federate` interface defines the simulator actions. The default implementation `DefaultFederate` composes local `SimObject` objects and a `FSSambassador` to coordinate data exchange with the federation. The federation interface itself is defined via the `fss.xml` federation object model (FOM) file and default connection settings are saved in the `connection.data` file. Figure 4 illustrates a simplified object class diagram of the toolkit execution classes.

A number of listener interfaces and associated events provide observer capabilities, i.e. for graphical user interfaces. For example, a `ConnectionEvent` notifies when a federate connects or disconnects from a federation. An `ExecutionControlEvent` notifies when a simulation is initializes, starts, stops, or terminates. An `ObjectChangeEvent` notifies when an object is discovered, removed, or updated, or an interaction is received. Finally, a `SimulationTimeEvent` notifies when the simulation time is advanced.

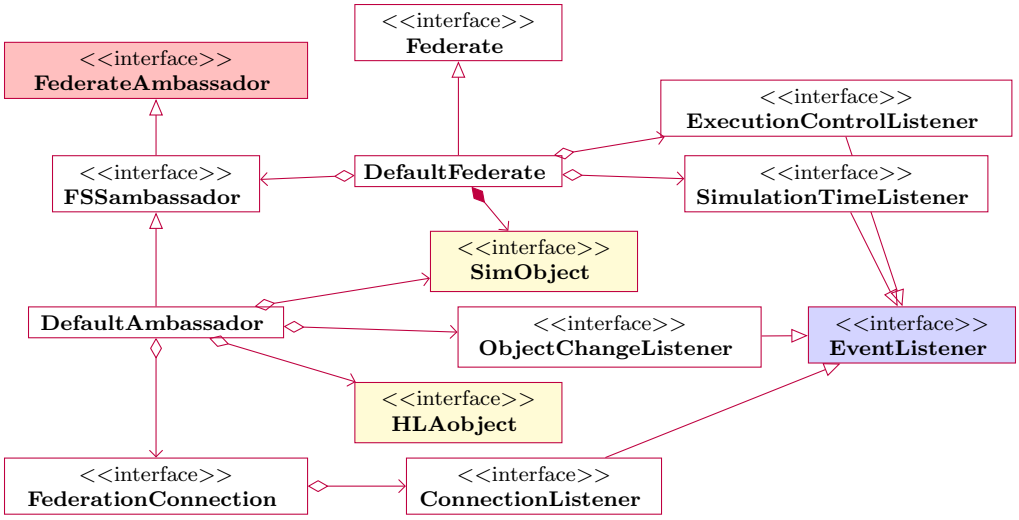

Figure 4: Object class diagram of the FSS simulation toolkit HLA model implementations

3.4 Toolkit Examples

The toolkit includes six example federates based on the case study in Grogan et al. (2014). Each federate includes a graphical user interface (GUI) for user interaction using common GUI components within the `edu.mit.fss.examples.gui` package. Five federates for FSS member simulation use customized data models within the `edu.mit.fss.examples.member` package and specialized GUI components within the `edu.mit.fss.examples.member.gui` package. A visualization federate uses the NASA World Wind application and specialized GUI components within the `edu.mit.fss.examples.visual.gui` package.

```
trunk/
└── src/
    ├── edu.mit.fss.examples
    │   ├── CosmoSkyMed1.java ..... CosmoSkyMED-1 FSS member federate example
    │   ├── ISSNode.java ..... ISS-Node FSS member federate example
    │   ├── SaudiComSat.java ..... SaudiComSat FSS member federate example
    │   ├── TDRSS.java ..... TDRSS FSS member federate example
    │   ├── TerraSarX.java ..... TerraSAR-X FSS member federate example
    │   └── Visualization.java ..... Visualization federate example
    ├── edu.mit.fss.examples.gui ..... Common GUI components for the examples
    ├── edu.mit.fss.examples.member ..... Model components for the FSS member example
    ├── edu.mit.fss.examples.member.gui ..... GUI components for the FSS member example
    └── edu.mit.fss.examples.visual.gui ..... GUI components for the visualization example
└── data.tle ..... Two line element data for example spacecraft
```

3.5 Toolkit Tutorials

The toolkit includes four short tutorials to introduce the approach for developing federates

```
trunk/
└── src/
    └── edu.mit.fss.tutorial.part1
        ├── ClockFederate.java ..... An example federate to simulate a clock object
        └── TrivialClock.java ..... A simple simulation object to keep track of time
```

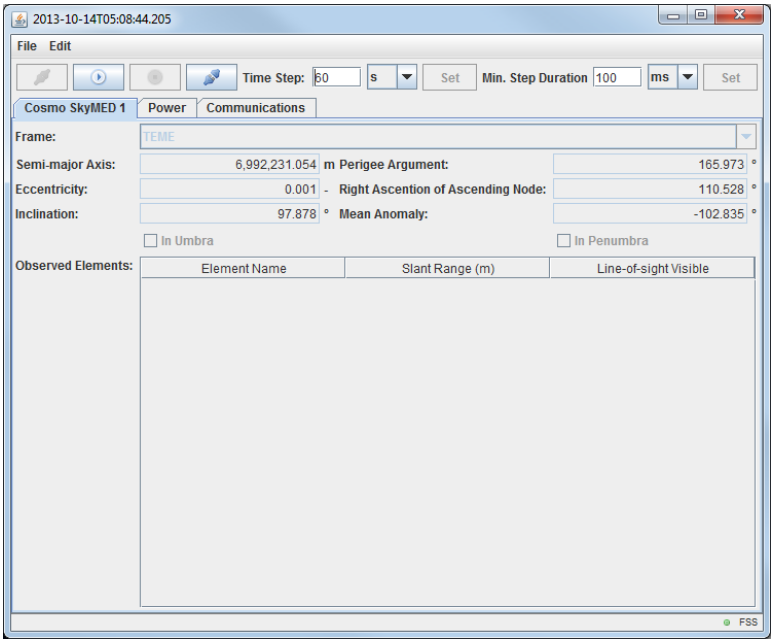

Figure 5: Screen capture of the CosmosSkyMED-1 example federate. The top component is an `ExecutionControlPanel`, below is a `SpaceSystemPanel` with tabs for an `OrbitalElementPanel`, `PowerSubsystemPanel`, and `CommSubsystemPanel`.

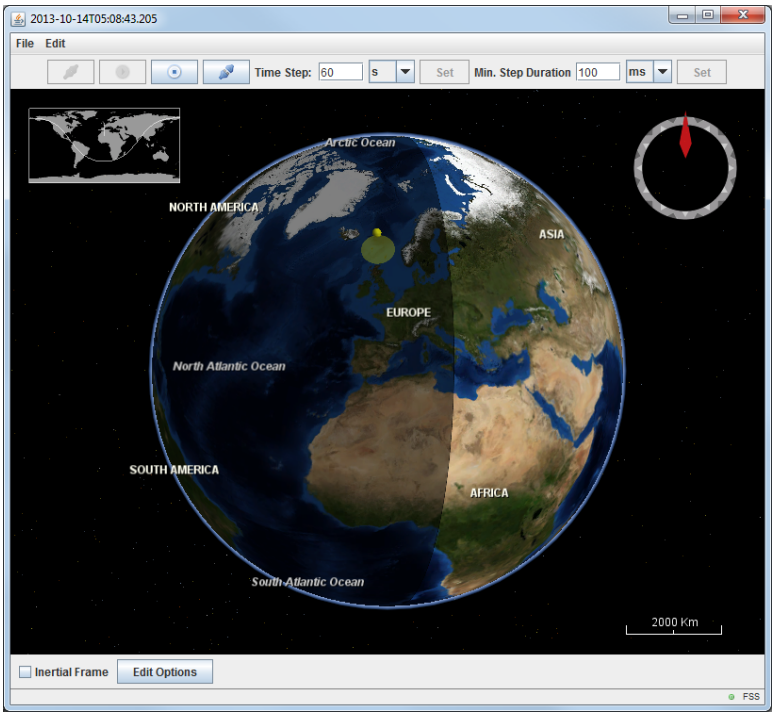

Figure 6: Screen capture of the example visualization federate. The top component is an `ExecutionControlPanel`, below is a `WorldWindVisualization`.

- edu.mit.fss.tutorial.part2
  - MobileElement.java ..... A simple SurfaceElement implementation
  - OfflineTutorialFederate.java .. An example federate to simulate a mobile element
- edu.mit.fss.tutorial.part3
  - Element1.java ..... Runs a federate for the "Element 1" mobile element
  - Element2.java ..... Runs a federate for the "Element 2" mobile element
  - OnlineTutorialFederate.java ... An example federate to simulate a mobile element
- edu.mit.fss.tutorial.part4
  - ControlPanel.java ..... A component to display and control a mobile element
  - ElementGUI.java ..... Runs a federate for a mobile element with a GUI

3.6 Third Party Libraries

The following third party libraries provide functionality for the toolkit:

- trunk/
  - lib/
    - commons-math3-3.2/ ..... Apache Commons Math 3 (Orekit dependency)
    - jfreechart-1.0.17/ ..... JFreeChart and JCommon for user interface charts
    - log4j-1.2.17/ ..... Java debugging and error logging
    - orekit-6.0/ ..... Orekit spacecraft flight dynamics
    - portico-2.0.0/ ..... Portico HLA runtime infrastructure
    - worldwind-2.0-857.1737/ ..... World Wind for Earth visualization
  - lib-external/
    - gdal/ ..... Geospatial Data Abstraction Library (World Wind dependency)
  - orekit-data.zip ..... Default physical data package for Orekit
  - RTI.rid ..... RTI Initialization Data (RID) file for Portico

In particular, the Portico library provides an open source HLA runtime infrastructure. Notably, it uses UDP multicast communication which may be blocked by some network infrastructure. Additional configuration settings for Portico are contained within the `RTI.rid` file.

The Orekit spacecraft flight dynamics library is used to define reference frames and to perform orbital propagation within the example federates. It requires a physical data package for which this toolkit uses the default file, `orekit-data.zip`.

The JFreeChart library is used within the example federates for charting and plotting capabilities.

The World Wind version distributed in this toolkit is a nightly-build dating to October 2013. It should be replaced with version 2.0 when officially released.

4 Quick Start Tutorials

This quick start tutorial develops new federates from the core FSS interfaces. It is intended to illustrate the concepts used to implement the example federates in the previous section through four successive parts.

Part 1 demonstrates how new objects can be defined and used in a time-stepped simulation. It implements a trivial clock simulation object and uses it to track time advancement in an offline simulation.

Part 2 demonstrates how the FSS core interfaces can be used to create a custom FSS object. It implements a mobile surface element with controllable velocity and uses it in an offline simulation.

Part 3 demonstrates an online distributed simulation using two instances of the mobile surface element implementation from Part 2. Two federates join a federated simulation and output computed distances from each other's element.

Finally, Part 4 develops a graphical user interface (GUI) to control the mobile surface elements from Part 3. The GUI allows velocity to be changed during a simulation using arrow key bindings.

4.1 Part 1: Implementing a Simulation Object

The first tutorial focuses on defining a very simple simulation object—a clock which only keeps track of time—and using it in an offline simulation. First, we define the `TrivialClock` object class implementing the `SimObject` interface described in Box 1. It contains data members for its name and time, and a temporary variable to hold the next time between `tick` and `tock` method calls. We also include a `getDate()` method to convert the integer-based time in milliseconds to a Java `Date` object.

Box 1: TrivialClock object class

```
1  package edu.mit.fss.tutorial.part1;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Date;
6  import edu.mit.fss.SimObject;
7
8  public class TrivialClock implements SimObject {
9      private String name;
10     private long time, nextTime;
11
12     public TrivialClock(String name) {
13         this.name = name;
14     }
15     public Date getDate() {
16         return new Date(time);
17     }
18     public String getName() {
19         return name;
20     }
21     public Collection<? extends SimObject> getNestedObjects() {
22         return new ArrayList<SimObject>();
23     }
24     public long getTime() {
25         return time;
26     }
27     public void initialize(long time) {
28         this.time = time;
29     }
30     public void tick(long duration) {
31         nextTime = time + duration;
32     }
33     public void tock() {
34         time = nextTime;
35     }
36 }
```

Next we define a new federate to use a `TrivialClock` object instance. The `ClockFederate` object class in Box 2 trivially extends the `DefaultFederate` class. Its main method runs the federate, performing the following actions:

1. Configures log messages.

2. Instantiates a new trivial clock object instance (`clock`).
3. Instantiates a new clock federate instance (`fed`).
4. Adds the clock instance to the federate.
5. Adds a `SimulationTimeListener` to the federate to log messages whenever the simulation time advances.
6. Sets the federate's initial (0 s) and final (20 s) times and the time step (1 s).
7. Configures the federate to execute in offline mode.
8. Initializes, runs, and finally exist the federate.

Box 2: ClockFederate object class

```java
package edu.mit.fss.tutorial.part1;

import hla.rti1516e.exceptions.RTIexception;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import edu.mit.fss.DefaultFederate;
import edu.mit.fss.event.SimulationTimeEvent;
import edu.mit.fss.event.SimulationTimeListener;

public class ClockFederate extends DefaultFederate {
    private static Logger logger = Logger.getLogger("edu.mit.fss");

    public ClockFederate() throws RTIexception {
        super();
    }
    public static void main(String[] args) throws RTIexception {
        BasicConfigurator.configure();
        logger.setLevel(Level.INFO);

        final TrivialClock clock = new TrivialClock("My clock");

        ClockFederate fed = new ClockFederate();
        fed.addObject(clock);
        fed.addSimulationTimeListener(new SimulationTimeListener() {
            public void timeAdvanced(SimulationTimeEvent event) {
                logger.info("The clock reads " + clock.getDate() + ".");
            }
        });
        fed.setInitialTime(0);
        fed.setFinalTime(20000);
        fed.setTimeStep(1000);
        fed.getConnection().setOfflineMode(true);
        fed.initialize();
        fed.run();
        fed.exit();
    }
}
```

A sample output from running the main method in `ClockFederate` is displayed in Box 3. The numbers on the left show the system time (in milliseconds) and the bracketed term identifies the thread executing the log comment (the main thread in this case). The default minimum step duration is quite visible in this case as each simulation time step is throttled to take at least 100 ms.

Box 3: ClockFederate main output

```
239 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:00 EST 1969.
243 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:01 EST 1969.
355 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:02 EST 1969.
455 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:03 EST 1969.
555 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:04 EST 1969.
655 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:05 EST 1969.
755 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:06 EST 1969.
855 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:07 EST 1969.
955 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:08 EST 1969.
1055 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:09 EST 1969.
1155 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:10 EST 1969.
1255 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:11 EST 1969.
1355 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:12 EST 1969.
1455 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:13 EST 1969.
1555 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:14 EST 1969.
1655 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:15 EST 1969.
1755 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:16 EST 1969.
1856 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:17 EST 1969.
1955 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:18 EST 1969.
2055 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:19 EST 1969.
2155 [main] INFO edu.mit.fss - The clock reads Wed Dec 31 19:00:20 EST 1969.
```

4.2 Part 2: Implementing an FSS Element

The second part of this tutorial implements an FSS element capable of participating in a federated simulation. For simplicity, we implement a `SurfaceElement` relying on a basic Earth-fixed Cartesian coordinate system (XY-plane on the equator, positive-X towards the prime meridian, positive-Y towards 90-degrees longitude, and positive-Z towards the North pole).

Box 4 describes the implementation of the `MobileElement` object class. It includes data members for its name, position, and velocity with additional data members to temporarily store the next and initial positions and velocities. The constructor takes a name and initial velocity as arguments and sets other variables to default values.

The `tick` method uses a simple Euler integration to propagate position based on an assumption of fixed velocity. A more sophisticated implementation may use a higher-fidelity integrator such as one from a selection of Runge-Kutta integrators included in the Apache Commons Math library.

The Geodetic coordinate implementations (i.e. latitude, longitude, and altitude) use an inaccurate spherical Earth model, i.e. they are computed using simple spherical coordinate transforms assuming a constant radius of 6371 kilometers rather than the standard WGS84 Earth model assumed for the ITRF2008 reference frame. The toolkit-provided `OrekitSurfaceElement`, for example, uses the WGS84 model parameters in the Orekit library to transform between Cartesian and Geodetic coordinates.

Box 4: MobileElement object class

```
package edu.mit.fss.tutorial.part2;

import java.util.ArrayList;
import java.util.Collection;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
import edu.mit.fss.ReferenceFrame;
import edu.mit.fss.SimObject;
import edu.mit.fss.SurfaceElement;

public class MobileElement implements SurfaceElement {
    private String name;
    private Vector3D position, nextPosition, initialPosition;
    private Vector3D velocity, nextVelocity, initialVelocity;

    public MobileElement(String name, Vector3D initialVelocity) {
        this.name = name;
        position = new Vector3D(0, 0, 0);
        initialPosition = new Vector3D(0, 0, 0);
        velocity = new Vector3D(0, 0, 0);
        this.initialVelocity = initialVelocity;
    }
    public double getAltitude() { return position.getNorm() - 6371e3; }
    public ReferenceFrame getFrame() { return ReferenceFrame.ITRF2008; }
    public double getLatitude() {
        return Math.toDegrees(Math.atan2(position.getZ(), position.getX()));
    }
    public double getLongitude() {
        return Math.toDegrees(Math.atan2(position.getY(), position.getX()));
    }
    public String getName() { return name; }
    public Collection<? extends SimObject> getNestedObjects() {
        return new ArrayList<SimObject>();
    }
    public Vector3D getPosition() { return position; }
    public Vector3D getVelocity() { return velocity; }
    public void initialize(long time) {
        position = initialPosition;
        nextPosition = initialPosition;
        velocity = initialVelocity;
        nextVelocity = initialVelocity;
    }
    public void setVelocity(Vector3D velocity) { nextVelocity = velocity; }
    public void tick(long duration) {
        nextPosition = position.add(velocity.scalarMultiply(duration/1000d));
    }
    public void tock() {
        position = nextPosition;
        velocity = nextVelocity;
    }
}
```

We will use the `MobileElement` implementation throughout the rest of the tutorial exercises. First, however, let's test in an offline federation. Box 5 extends the federate developed in Part 1 to include a new mobile element. Note that as it is initialized to the center of the Earth (0,0,0), there is no physical interpretation for this particular element.

Box 5: OfflineTutorialFederate object class

```
1  package edu.mit.fss.tutorial.part2;
2
3  import hla.rti1516e.exceptions.RTIexception;
4  import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
5  import org.apache.log4j.BasicConfigurator;
6  import org.apache.log4j.Level;
7  import org.apache.log4j.Logger;
8  import edu.mit.fss.DefaultFederate;
9  import edu.mit.fss.event.SimulationTimeEvent;
10 import edu.mit.fss.event.SimulationTimeListener;
11
12 public class OfflineTutorialFederate extends DefaultFederate {
13     private static Logger logger = Logger.getLogger("edu.mit.fss");
14
15     public OfflineTutorialFederate(final MobileElement element) throws RTIexception {
16         super();
17         addObject(element);
18         addSimulationTimeListener(new SimulationTimeListener() {
19             public void timeAdvanced(SimulationTimeEvent event) {
20                 logger.info("At time " + event.getTime() + ", "
21                         + element.getName() + " is at "
22                         + element.getPosition() + ".");
23             }
24         });
25     }
26     public void execute(long initialTime, long finalTime, long timeStep) {
27         setInitialTime(initialTime);
28         setFinalTime(finalTime);
29         setTimeStep(timeStep);
30         getConnection().setOfflineMode(true);
31         initialize();
32         run();
33     }
34     public static void main(String[] args) throws RTIexception {
35         BasicConfigurator.configure();
36         logger.setLevel(Level.INFO);
37         MobileElement element = new MobileElement("Element", new Vector3D(10, 0, 0));
38         OfflineTutorialFederate fed = new OfflineTutorialFederate(element);
39         fed.execute(0, 20000, 1000);
40         fed.exit();
41     }
42 }
```

The `OfflineTutorialFederate` includes more internal structure than `ClockFederate`. In particular, several instructions have been moved to the constructor and others to a new `execute` method. The constructor adds the mobile element to the federate and adds a `SimulationTimeListener` to

log messages during the simulation execution. The `execute` method sets the initial and final time, the time step, configures the federate for offline simulation, and initializes and runs the federate.

The main method contains other instructions, such as configuring logging, specifying the mobile element object instantiation (with initial velocity of 10 m/s in the X-direction) and federate object instantiation. It also defines the execution period between 0 and 20 seconds with a 1-second time step. After the federate executes, it exits.

Box 6 displays a sample output from the main method. The element moves 10 meters in the x-direction during each 1-second time step according to its constant velocity.

Box 6: OfflineTutorialFederate main output

```
218 [main] INFO edu.mit.fss - At time 0, Element is at {0; 0; 0}.
221 [main] INFO edu.mit.fss - At time 1000, Element is at {10; 0; 0}.
330 [main] INFO edu.mit.fss - At time 2000, Element is at {20; 0; 0}.
430 [main] INFO edu.mit.fss - At time 3000, Element is at {30; 0; 0}.
530 [main] INFO edu.mit.fss - At time 4000, Element is at {40; 0; 0}.
630 [main] INFO edu.mit.fss - At time 5000, Element is at {50; 0; 0}.
730 [main] INFO edu.mit.fss - At time 6000, Element is at {60; 0; 0}.
830 [main] INFO edu.mit.fss - At time 7000, Element is at {70; 0; 0}.
943 [main] INFO edu.mit.fss - At time 8000, Element is at {80; 0; 0}.
1043 [main] INFO edu.mit.fss - At time 9000, Element is at {90; 0; 0}.
1143 [main] INFO edu.mit.fss - At time 10000, Element is at {100; 0; 0}.
1243 [main] INFO edu.mit.fss - At time 11000, Element is at {110; 0; 0}.
1343 [main] INFO edu.mit.fss - At time 12000, Element is at {120; 0; 0}.
1444 [main] INFO edu.mit.fss - At time 13000, Element is at {130; 0; 0}.
1543 [main] INFO edu.mit.fss - At time 14000, Element is at {140; 0; 0}.
1643 [main] INFO edu.mit.fss - At time 15000, Element is at {150; 0; 0}.
1743 [main] INFO edu.mit.fss - At time 16000, Element is at {160; 0; 0}.
1843 [main] INFO edu.mit.fss - At time 17000, Element is at {170; 0; 0}.
1944 [main] INFO edu.mit.fss - At time 18000, Element is at {180; 0; 0}.
2043 [main] INFO edu.mit.fss - At time 19000, Element is at {190; 0; 0}.
2143 [main] INFO edu.mit.fss - At time 20000, Element is at {200; 0; 0}.
```

4.3 Part 3: Running a Federated Simulation

The third part of this tutorial uses the `MobileElement` object developed in Part 2 in an online federated simulation. Box 7 implements the `OnlineTutorialFederate` as an extension of the offline federate developed in Part 2 with several additions.

First, two new data members, `thisElement` and `otherElements` are added to track the local and remote objects respectively. The `otherElements` object uses a synchronized list for thread safety as the HLA uses a separate thread for object updates. The constructor adds a synchronized loop to print information for the `otherElements` within the `SimulationTimeListener` and adds an `ObjectChangeListener` to add or remove objects from the `otherElements` list. Finally, the `execute` method configures the federate name, type, federation name, and FOM path.

Box 7: OnlineTutorialFederate object class

```
package edu.mit.fss.tutorial.part3;
import hla.rti1516e.exceptions.RTIexception;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import org.apache.log4j.Logger;
import edu.mit.fss.DefaultFederate;
import edu.mit.fss.SurfaceElement;
import edu.mit.fss.event.*;
import edu.mit.fss.tutorial.part2.MobileElement;
public class OnlineTutorialFederate extends DefaultFederate {
    private static Logger logger = Logger.getLogger(OnlineTutorialFederate.class);
    private MobileElement thisElement;
    private List<SurfaceElement> otherElements = Collections.synchronizedList(
                new ArrayList<SurfaceElement>());
    public OnlineTutorialFederate(MobileElement element) throws RTIexception {
        super();
        thisElement = element; addObject(element);
        addSimulationTimeListener(new SimulationTimeListener() {
            public void timeAdvanced(SimulationTimeEvent event) {
                logger.info("At time " + event.getTime() + ", " + thisElement.getName()
                        + " is at " + thisElement.getPosition() + ".");
                synchronized(otherElements) {
                    for(SurfaceElement e : otherElements) {
                        logger.info("... distance to " + e.getName() + " is "
                                + thisElement.getPosition().distance(e.getPosition()) + ".");
                    }
                }
            }
        });
        addObjectChangeListener(new ObjectChangeListener() {
            public void objectDiscovered(ObjectChangeEvent event) {
                if(event.getObject() instanceof SurfaceElement
                        && event.getObject() != thisElement) {
                    otherElements.add((SurfaceElement) event.getObject());
                }
            }
            public void objectRemoved(ObjectChangeEvent event) {
                otherElements.remove(event.getObject());
            }
            public void objectChanged(ObjectChangeEvent event) {  }
            public void interactionOccurred(ObjectChangeEvent event) {  }
        });
    }
    public void execute(long initialTime, long finalTime, long timeStep) {
        setInitialTime(initialTime); setFinalTime(finalTime); setTimeStep(timeStep);
        getConnection().setOfflineMode(false);
        getConnection().setFederateName(thisElement.getName());
        getConnection().setFederateType("Demo");
        getConnection().setFederationName("Tutorial");
        getConnection().setFomPath("fss.xml");
        connect(); initialize(); run();
    }
}
```

Boxes 8 and 9 create main methods for two federates. They use a similar form with differences in element name (Element 1 and Element 2) and initial velocity (-10 m/s in the X-direction and +5 m/s in the Y-direction).

Box 8: Element1 object class

```
package edu.mit.fss.tutorial.part3;

import hla.rti1516e.exceptions.RTIexception;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import edu.mit.fss.tutorial.part2.MobileElement;

public abstract class Element1 {
    private static Logger logger = Logger.getLogger("edu.mit.fss");

    public static void main(String[] args) throws RTIexception {
        BasicConfigurator.configure();
        logger.setLevel(Level.INFO);

        MobileElement element = new MobileElement("Element 1", new Vector3D(10, 0, 0));
        OnlineTutorialFederate fed = new OnlineTutorialFederate(element);
        fed.execute(0, 50000, 1000);
        fed.exit();
    }
}
```

Box 9: Element2 object class

```
package edu.mit.fss.tutorial.part3;

import hla.rti1516e.exceptions.RTIexception;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import edu.mit.fss.tutorial.part2.MobileElement;

public abstract class Element2 {
    private static Logger logger = Logger.getLogger("edu.mit.fss");

    public static void main(String[] args) throws RTIexception {
        BasicConfigurator.configure();
        logger.setLevel(Level.INFO);

        MobileElement element = new MobileElement("Element 2", new Vector3D(0, 5, 0));
        OnlineTutorialFederate fed = new OnlineTutorialFederate(element);
        fed.execute(0, 50000, 1000);
        fed.exit();
    }
}
```

Boxes 10 and 11 illustrate output from the two federates. In this particular case, the Element1 federate starts first and the Element2 federate joins the federation at simulation time 15000. Note the differences between the main thread (`[main]`) and the other thread (`[Thread-0]`) in logging statements. Also note the relative delay in updating element attributes: the Element2 name is not received by Element1 until 16000 and the Element1 name is not received by Element2 until 17000. The timing of all of these events is dependent on network latency, processor utilization, and other operating system factors not easily controllable without additional HLA services such as synchronization points.

Box 10: Element1 partial main output

```
13674 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 13000.0
13675 [main] INFO edu.mit.fss... - At time 13000, Element 1 is at {130; 0; 0}.
13788 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 14000.0
13788 [main] INFO edu.mit.fss... - At time 14000, Element 1 is at {140; 0; 0}.
13851 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 15000.0
13851 [main] INFO edu.mit.fss... - At time 15000, Element 1 is at {150; 0; 0}.
15080 [Thread-0] INFO edu.mit.fss... - Discovering object instance 2097153.
15095 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 16000.0
15095 [main] INFO edu.mit.fss... - At time 16000, Element 1 is at {160; 0; 0}.
15095 [main] INFO edu.mit.fss... - ... distance to Element 2 is 160.0781059358212.
15158 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 17000.0
15158 [main] INFO edu.mit.fss... - At time 17000, Element 1 is at {170; 0; 0}.
15159 [main] INFO edu.mit.fss... - ... distance to Element 2 is 170.29386365926402.
```

Box 11: Element2 partial main output

```
10232 [main] INFO edu.mit.fss... - Joined federation execution Tutorial as federate
    Element 2 of type Demo.
10240 [main] INFO edu.mit.fss... - Asynchronous message delivery enabled.
10247 [Thread-0] INFO edu.mit.fss... - Time constrained enabled with logical time 0.0.
10247 [main] INFO edu.mit.fss... - Time constrained behavior enabled.
10251 [Thread-0] INFO edu.mit.fss... - Time regulation enabled with logical time 15000.0.
11250 [main] INFO edu.mit.fss... - Time regulating behavior enabled.
11270 [Thread-0] INFO edu.mit.fss... - Discovering object instance 2.
11378 [main] INFO edu.mit.fss... - Published and subscribed all objects and interactions.
11384 [main] INFO edu.mit.fss... - At time 15000, Element 2 is at {0; 0; 0}.
11396 [main] INFO edu.mit.fss... - ... distance to is 0.0.
11426 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 16000.0
11426 [main] INFO edu.mit.fss... - At time 16000, Element 2 is at {0; 5; 0}.
11426 [main] INFO edu.mit.fss... - ... distance to is 5.0.
11503 [Thread-0] INFO edu.mit.fss... - Time advance granted to logical time 17000.0
11505 [main] INFO edu.mit.fss... - At time 17000, Element 2 is at {0; 10; 0}.
11506 [main] INFO edu.mit.fss... - ... distance to Element 1 is 170.29386365926402.
```

4.4 Part 4: Creating a Graphical User Interface

The fourth part of this tutorial builds a graphical user interface to control a `MobileElement` object as a federate. Boxes 12 and 13 develop a `ControlPanel` object class which displays the spatial state of a federate in a graphical user interface.

Similar to the `OnlineTutorialFederate`, it contains data members `boundElement` and `elements` for a local element and list of remote elements (again using a synchronized list). A `KeyAdapter` object added in the constructor responds to key presses to control the velocity of the bound element. Methods from the `ObjectChangeListener` interface add or remove elements or prompt a repainting of the display. Finally, the `paint` method uses a custom method to update the display, using lines for x- and y-axes and circles (with labels) for each element. The bound element is drawn in blue while other elements are drawn in black.

Box 12: ControlPanel object class

```
package edu.mit.fss.tutorial.part4;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import javax.swing.JPanel;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
import edu.mit.fss.SurfaceElement;
import edu.mit.fss.event.ObjectChangeEvent;
import edu.mit.fss.event.ObjectChangeListener;
import edu.mit.fss.tutorial.part2.MobileElement;

public class ControlPanel extends JPanel implements ObjectChangeListener {
    private static final long serialVersionUID = -1729012305362962099L;
    private static float DISPLAY_SCALE = 5.0f;
    private static int ELEMENT_SIZE = 4;
    private MobileElement boundElement;
    private List<SurfaceElement> elements = Collections.synchronizedList(new
        ArrayList<SurfaceElement>());
    public ControlPanel() {
        setFocusable(true);
        setPreferredSize(new Dimension(300,300));
        setBackground(Color.white);
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if(boundElement==null) { return; }
                if(e.getKeyCode()==KeyEvent.VK_UP) {
                    boundElement.setVelocity(boundElement.getVelocity().add(new
                        Vector3D(0,1,0)));
                } else if(e.getKeyCode()==KeyEvent.VK_DOWN) {
                    boundElement.setVelocity(boundElement.getVelocity().add(new
                        Vector3D(0,-1,0)));
                } else if(e.getKeyCode()==KeyEvent.VK_LEFT) {
                    boundElement.setVelocity(boundElement.getVelocity().add(new
                        Vector3D(-1,0,0)));
                } else if(e.getKeyCode()==KeyEvent.VK_RIGHT) {
                    boundElement.setVelocity(boundElement.getVelocity().add(new
                        Vector3D(1,0,0)));
                }
            }
        });
    }
    private int[] getScreenLocation(Vector3D position) {
        int x = (int) Math.round(getWidth()/2 + position.getX()/DISPLAY_SCALE);
        int y = (int) Math.round(getHeight()/2 - position.getY()/DISPLAY_SCALE);
        return new int[]{x, y};
    }
    public void interactionOccurred(ObjectChangeEvent event) { }
```

Box 13: ControlPanel object class (continued)

```
    public void objectChanged(ObjectChangeEvent event) {
        repaint();
    }
    public void objectDiscovered(ObjectChangeEvent event) {
        if(event.getObject() instanceof SurfaceElement) {
            elements.add((SurfaceElement) event.getObject());
        }
    }
    public void objectRemoved(ObjectChangeEvent event) {
        elements.remove(event.getObject());
    }
    public void paint(Graphics g) {
        super.paint(g);
        g.setColor(Color.gray);
        g.drawLine(0, getHeight()/2, getWidth(), getHeight()/2);
        g.drawLine(getWidth()/2, 0, getWidth()/2, getHeight());
        synchronized(elements) {
            for(SurfaceElement e : elements) {
                if(e instanceof MobileElement) {
                    g.setColor(Color.blue);
                } else {
                    g.setColor(Color.black);
                }
                int[] location = getScreenLocation(e.getPosition());
                g.fillOval(location[0] - ELEMENT_SIZE/2,
                        location[1] - ELEMENT_SIZE/2,
                        ELEMENT_SIZE, ELEMENT_SIZE);
                g.drawString(e.getName(), location[0] + ELEMENT_SIZE/2 + 2,
                        location[1] + ELEMENT_SIZE/2 + ELEMENT_SIZE/2);
            }
        }
    }
    public void setBoundElement(MobileElement element) {
        this.boundElement = element;
    }
}
```

Box 14 develops the `ElementGUI` class which holds the main method to run the federate. It uses a `JOptionPane` to allow user input for a name of the local mobile element instance. Finally it creates a `ControlPanel` object instance, adds it as an `ObjectChangeListener` to an `OnlineTutorialFederate`, and launches the GUI in a `JFrame`.

Box 14: ElementGUI object class

```java
package edu.mit.fss.tutorial.part4;

import hla.rti1516e.exceptions.RTIexception;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.lang.reflect.InvocationTargetException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import edu.mit.fss.tutorial.part2.MobileElement;
import edu.mit.fss.tutorial.part3.OnlineTutorialFederate;

public abstract class ElementGUI {
    private static Logger logger = Logger.getLogger("edu.mit.fss");

    public static void main(String[] args) throws RTIexception {
        BasicConfigurator.configure();
        logger.setLevel(Level.INFO);
        String name = null;
        while(name == null || name.isEmpty()) {
            name = JOptionPane.showInputDialog("Enter element name:");
        }
        final MobileElement element = new MobileElement(name, new Vector3D(0, 0, 0));
        final OnlineTutorialFederate fed = new OnlineTutorialFederate(element);
        try {
            SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                    ControlPanel controlPanel = new ControlPanel();
                    controlPanel.setBoundElement(element);
                    fed.addObjectChangeListener(controlPanel);
                    JFrame frame = new JFrame();
                    frame.setContentPane(controlPanel);
                    frame.pack();
                    frame.setVisible(true);
                    frame.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                            fed.exit();
                        }
                    });
                }
            });
        } catch (InvocationTargetException | InterruptedException e) {
            logger.error(e);
        }
        fed.execute(0, Long.MAX_VALUE, 1000);
    }
}
```

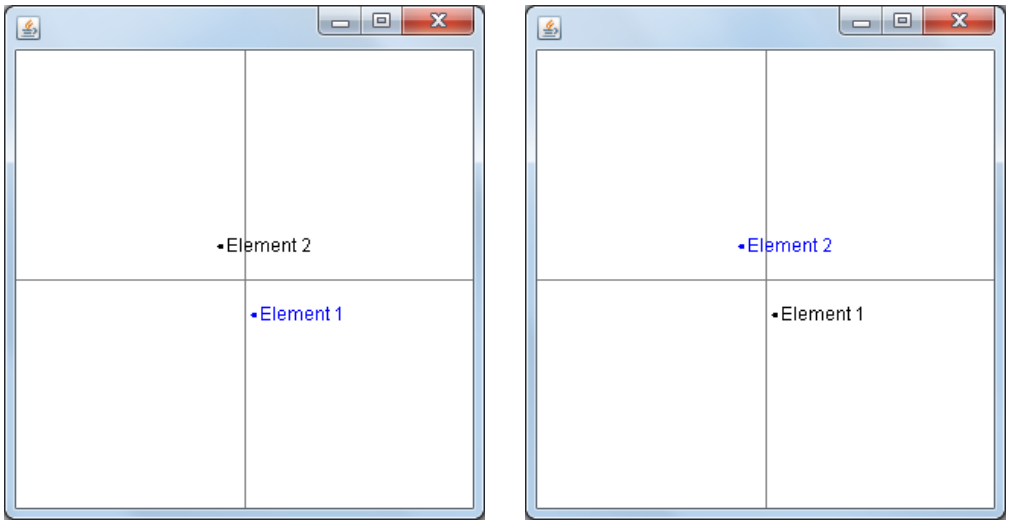

Figure 7: Screen capture of two `ElementGUI` instances. Left: bound element is named "Element 1." Right: bound element is named "Element 2."

Figure 7 shows screen captures of two instances of a `ElementGUI`. They display both elements on the X-Y plane, using blue for the local element with velocity controllable with arrow keys and black for the remote element.