A Coordinated Initialization Process for the Distributed Space Exploration Simulation (DSES)

Robert Phillips, Dan Dexter and David Hasan
L3 Communications, Inc.
1002 Gemini Avenue, Suite 200
Houston, TX 77058
(281) 483-0926, (281) 483-1142, (281) 480-4101
robert.g.phillips@L-3Com.com, dan.dexter@L-3Com.com, david.hasan@L-3Com.com

Edwin Z. Crues, Ph.D.
Simulation and Graphics Branch (ER7)
Automation, Robotics and Simulation Division
NASA Johnson Space Center
2101 NASA Parkway
Houston, TX 77058
281-483-2902
edwin.z.crues@nasa.gov

Keywords:
Distributed Simulation, Space, Exploration, NASA

ABSTRACT: *This document describes the federate initialization process that was developed at the NASA Johnson Space Center with the HIIA Transfer Vehicle Flight Controller Trainer (HTV FCT) simulations and refined in the Distributed Space Exploration Simulation (DSES). These simulations use the High Level Architecture (HLA) IEEE 1516 to provide the communication and coordination between the distributed parts of the simulation. The purpose of the paper is to describe a generic initialization sequence that can be used to create a federate that can:*

1. *Properly initialize all HLA objects, object instances, interactions, and time management*
2. *Check for the presence of all federates*
3. *Coordinate startup with other federates*
4. *Robustly initialize and share initial object instance data with other federates.*

1 Introduction

This document describes the federate initialization process that was developed at the NASA Johnson Space Center (JSC) with the HIIA Transfer Vehicle Flight Controller Trainer (HTV FCT) simulation and refined in the Distributed Space Exploration Simulation (DSES) [1][2]. These simulations use the High Level Architecture (HLA) IEEE 1516 to provide the communication and coordination between the distributed parts of the simulation (federates) [3]. This document assumes a basic understanding of the HLA IEEE 1516 standard, and C++ programming.

2 Background

The HTV FCT simulation was the first distributed simulation developed at JSC that used the IEEE 1516 standard. After the original concept had been proven, engineers at JSC started adding generic IEEE 1516 capabilities to the Trick simulation development package.

Engineers at JSC then worked with other NASA engineers at Ames Research Center, Langley Research Center, and Marshall Spaceflight Center to test some distributed simulation ideas for the Crew Exploration Vehicle (CEV) and Constellation programs in the DSES simulation.

It became clear during initial discussions about the DSES simulation that certain features would be important for the proposed environment. In particular, the ability to dynamically configure which vehicles (each represented by a federate) participate in a given run and the ability to dynamically share the initial configuration of each participating vehicle would both be very useful.

A generalized initialization routine usable by all federates was developed and tested that ensured a robust simulation run start regardless of participating federates or startup order and also allowed generic exchange of initial data. Since the other centers were not then using Trick and were not familiar with IEEE 1516, previous versions of this paper served as a "how to get started" document.

3 The Initialization Process

The initialization sequence described in this paper consists of the following steps, which are executed by each participating federate.

1. Create the federation
2. Publish and subscribe
3. Create object instances
4. Confirm all federates are joined
5. Achieve "initialize" Synchronization Point
6. Update object instance(s) with initial data
7. Wait for object instance reflections
8. Set up time management
9. Achieve "startup" Synchronization Point

3.1 Create the Federation

In this step, each participant in the simulation ensures that the federation exists and then joins it.

Every federate attempts to create the federation and, of course, only the first attempt will succeed. Subsequent attempts will generate an exception indicating that the federation already exists, which is not a failure condition but rather indicates that some other federate already created it. Following that, the federate joins the federation. (See Code Example 1.)

3.2 Publish and Subscribe

In this step, each federate publishes[1] and subscribes all relevant objects and interactions.

Publishing and subscribing for objects is done on an attribute-by-attribute basis. Consequently, federates must specify each object and its attributes of interest. On the other hand, publishing and subscribing to interactions includes all parameters. Consequently, federates only specify the interaction when publishing or subscribing to interactions. (See Code Examples 2 and 3.)

3.3 Create Object Instances

In this step, each federate creates instances[2] of the objects attributes of which it intends to publish.

Although it is not strictly necessary for the federate to immediately create instances of objects that are neither present at startup nor used during initialization, the DSES approach is to consolidate all instance creation code in one place. In any event, all objects used during initialization must be created at this point.

Instance names. It is possible that different federates will create object instances for the same object. Two instances of the object, `VehicleState`, could exist, for example, for the space station and crew exploration vehicle. This means that a federate that subscribes to an object might discover multiple instances and must be able to distinguish between them. To provide for this, IEEE 1516 allows federates to tag object instances with federation-unique names.

The DSES naming convention uses the space vehicle component names as a kind of de facto namespace for the object instance names. DSES vehicle names are used as name prefixes: `CLV_S1`, `CLV_S2`, `CEV_SM`, `CEV_CM`, and `LAS` (for Crew Launch Vehicle Stage 1 and Stage 2, CEV Service and Command Modules, and Launch Abort System).

A more general naming convention is to use a combination of the federate name and the object name. For example, if a federate publishes only one instance of a given object, it could name the instance `<federate-name>_<object-name>`. If a federate publishes one or more instances of a given object, it could name the instances `<federate-name>_<object-name>_<instance-name>`.

For example, if federate named `F1` publishes an object named `VehicleState`, it could name the object instance `F1_VehicleState`. If the federate needs to

[1] In HLA, the term *publish* refers to a federate's announcement to the RTI of its intention to create and update object attributes or interactions. No data is actually sent during publishing.

[2] In HLA, an *object* is a like a Java/C++ type or class, and an *instance* is a specific instantiation of that type/class.

create two instances of that object for vehicles A and B, it could use names `F1_VehicleState_A` and `F1_VehicleState_B`.

Registering instance names. To assign names to object instances federates reserve the names with the Runtime Infrastructure (RTI) and then wait for confirmation. For the reserved name to be valid, it must be unique in the federation and not start with the prefix `HLA`. (See Code Example 4.) In response, the RTI invokes a federate ambassador callback indicating whether or not the reservation request succeeded. (See Code Example 5.) Once the reservation has been confirmed by the RTI, the federate may register an instance of the object with that name. (See Code Example 6.) Finally, after a publishing federate has registered an object instance, the RTI will invoke a federate ambassador callback for each subscribing federate to notify them of the new instance. (See Code Example 7.)

3.4 Confirm All Federates Are Joined

In this step, each federate must wait until all the expected federates have joined the federation.

Each federate could execute logic to ensure that all the other federates have joined, but DSES uses a mechanism in which only *one* federate does this. The approach uses two synchronization points*:* one called `initialize` that marks when all federates are ready to exchange initialization information and one called `startup` that marks when all federates have completed initialization and are ready to execute[3].

Registering synchronization points. Each federate attempt to register both synchronization points. (See Code Example 8.) In response, the RTI invokes federate ambassador callbacks indicating whether or not the registration request succeeded. Since only one federate can succeed in registering each synchronization point, all but one receive an "already exists" indication from the failure callback. This is not an error, rather it indicates that one of the other federates already registered the synchronization point. The federate that successfully registers the `initialize` synchronization point becomes the *master federate*[4]. (See Code Example 9.) All federates wait for both synchronization points to be registered before proceeding. (See Code Example 10.)

Waiting for joiners. At this point, the federates wait for all of the expected federates to join the federation. The DSES approach is to have the master federate do the work while the others wait for the master. The non-masters wait for the master at the `initialize` synchronization point. (See Code Example 11.)

In order to detect joined federates, the master federate uses an object from the Management Object Model (MOM) interface[5]. This MOM federate object has an attribute that gets updated as each new federate joins the federation, and the master federate subscribes to this attribute. (See Code Example 12.) The RTI invokes callbacks when this object is discovered and its attribute reflected. The value of the newly reflected attribute holds the name of a federate that has just joined the federation. The callback records this name in a data structure that lists all joined federates. (See Code Example 13.)

In DSES, every federate is configured with a run-time list of names of all the expected federates. The master federate repeatedly compares the list of expected federates to the list of joined federates until all the expected federates have joined. (See Code Example 14.)

This approach is crucial, because it allows the simulation to be easily started across multiple locations without a required startup order, and allows the set of participating federates to be easily modified before a run by editing the list of expected federates.

3.5 Achieve "initialize" Synchronization Point

In this step, each federate waits for the master federate to determine that all the expected federates have indeed joined the federation and that all other federates are ready to exchange initial data.

When the master federate reaches this step, it has assured that all expected federates have joined the federation. Furthermore, each federate upon reaching this step is completely ready to exchange initial data, because all required objects and interactions have been published and subscribed to, and all object instances have been registered.

---

[3] The `startup` synchronization point is actually not used until later; however, DSES initializes both the synchronization points together.

[4] Any federate might end up being the master.

[5] This object instance is published and reflected by the RTI itself (not by one of the federates).

Each federate achieves the `initialize` synchronization point to indicate that it is ready to exchange initial data, then enters a loop to wait for the `federationSynchronized` callback from the RTI. (See Code Example 15.) This callback is invoked when all federates have achieved the synchronization point, so each federate knows that the entire federation is present and ready to exchange initial data. (See Code Example 16.)

3.6 Update Object Instances With Initial Data

In this step, the federates exchange initial data. Each publishing federate reflects the initial values for all the attributes it owns, and these initial data are delivered to all subscribers.

There must be some agreement between federate implementers about which object instances are initialized statically by each federate and which will have initial data sent at run time by the owning federate. DSES federates exchange initialization data for *all* object instances at run time, because this avoids the configuration management issues associated with consistently duplicating static initialization data at the various federate locations. If instead, for example, static initialization data is read from input files, then each federate must have an input file that contains the initial values for the data published by the other federates.

Before sending initial data, DSES federates enable asynchronous delivery of Receive Order object instance reflections. Without this call, the federate will not receive the initialization data until it requests a time advance. Since the simulations have not started at this point, time ordered delivery is meaningless. The initial values may be set in any order, as long as they all get set. Enabling asynchronous delivery is needed so that data reflections can be received while the federate is paused.

After the federate has enabled asynchronous delivery, it sends the updates for the desired object instances. (See Code Example 17.)

All of the data in the DSES simulation is defined as Time Stamp Ordered (TSO). Since the initial data is sent before the federates have enabled time regulation and time constraints, it will be delivered as Receive Order (RO). Even if time regulation and constraints had been activated, the initial data could be sent as RO by using the `updateAttributeValues()` call without a time stamp.

3.7 Wait for Object Instance Reflections

In this step, each federate executes a wait loop until the expected initial object instance reflections from all other federates have been received. (See Code Example 18.)

In the Federate Ambassador, something like Code Example 19 will store reflected initialization values from the other federates and check to see if all federates have updated data for their initialization object instances.

3.8 Set Up Time Management

In this step, the federates activate time management.

3.8.1 Time Frames

Time management defines how the RTI synchronizes and relates the time for various federates. To understand this, it is important to keep in mind the distinction between RTI time, real time, and simulation time.

Most simulations of time-propagated dynamical systems have natural simulation time scales. However, this is typically not the case for either distributed or real-time simulations. In fact, a typical IEEE 1516 federate must operate in several distinct time frames simultaneously.

Real Time. This is the computer's concept of time passage in the physical world. This most often ties to registers in the computer's hardware that store values incremented in conjunction with an oscillator of known frequency and fidelity. These values can then be translated into a current time. In some cases, external interrupts or external clock registers are used.

Real time is sometimes referred to as *wall clock time*. It is important when a simulation is interfacing with time critical hardware or software or has elements that provide or require human interaction.

Simulation Time. This is the natural time scale for the dynamic systems being simulated. From a simulation's (and therefore a federate's) point of view, this is its "actual" time -- the time that it is currently simulating.

Simulation time advancement is determined by the needs of the dynamic system being simulated. For instance, Trick [4][5] based orbital dynamics simulations are often propagated in 0.01-second time steps (100 Hz). Trick based robotic simulations, however, are often propagated at 0.001-second time steps (1000 Hz).

In certain circumstances, the simulation time can keep up with or go faster than wall clock time. When simulation runs at the same rate as real time, the simulation is said to *run real time*.

RTI Time. This is the time that the RTI thinks the federate is at and therefore is the timetag associated with data reflected by the RTI. Since the RTI time can advance at a different rate (1Hz for all federates in the HTV FCT federation, 4Hz for all federates in the DSES federation) than the simulation time and the RTI time advances involve asynchronous callbacks from the RTI, the RTI time and simulation time are only loosely coupled.

Update Time. This is the earliest time for which the federate is allowed to publish. It is identical to the federate's Greatest Allowed Logical Time, or GALT. This is related to the RTI time as follows: if the federate is not in Time Advancing mode, the update time is equal to the federate's current RTI time plus its *lookahead time interval*. If the federate is in Time Advancing mode, then the update time is equal to the RTI time that the federate is advancing to plus its lookahead time interval. A federate is in Time Advancing mode after it has made a Time Advance Request or Time Advance Request Available, but before the corresponding Time Advance Grant has been received.

3.8.2 Time Management

In both the HTV FCT and the DSES federations, the default mode of operation is to have all federates be both time regulating and time constrained [6]. The time regulation uses a lookahead interval the same as the rate of data sending, e.g., 4Hz sending with 0.25 second lookahead. (See Code Example 20.) HLA does not require this, but it simplifies things to do so.

3.9 Achieve "startup" Synchronization Point

In this step, the federates synchronize at the `startup` synchronization point.

When all federates have done this and the federation has been synchronized, then the DSES initialization is complete and the federates can begin running the simulation. (See Code Examples 21 and 22.)

4 Concluding Remarks

It is clear that far more than the Federation Object model (FOM) and Simulation Object Model (SOM) are necessary to coordinate federates in an IEEE 1516 federation. This paper has presented an approach to standardizing federation initialization that addresses several general issues.

- It ensures that all necessary federates are joined before the simulation starts.
- It allows the set of necessary federates to be easily modified.
- It doesn't force a specific federate start sequence.
- It allows dynamic exchange of initial object attribute values during startup.

Simulations like DSES have multiple federates that represent separate systems with many possible configurations. The ability to ensure robust dynamic initialization without centralized control is crucial in this kind of environment.

5 References

[1] G. Lauderdale, E. Crues, D. Snyder, D. Hasan, "A Feasibility Study for ISS and HTV Distributed Simulation," AIAA Modeling and Simulation Technologies Conference and Exhibit, Austin, Texas, 11-14 August 2003.

[2] G. Lauderdale, E. Crues, D. Snyder, D. Hasan, "Further Studies On The Feasibility Of A Distributed ISS and HTV Simulation," Proceedings of the Fall 2003 Simulation Interoperability Workshop and Conference, Orlando, Florida, 15-18 September 2003.

[3] Simulation Interoperability Standards Committee (SISC) of the IEEE Computer Society, "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Federate Interface Specification", The Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, USA, 9 March 2001, ISBN 0-7381-2622-5 SS94883.

[4] K. Vetter, "Trick User's Guide 2005 Release," NASA Technical Publication, NASA Johnson Space Center, September 2005.

[5] K. Vetter, "Trick Simulation Environment – User Training Material 2005 Release," NASA Technical Publication, NASA Johnson Space Center, September 2005.eren

[6] R. Phillips, E. Crues, "Time Management Issues and Approaches For Real Time HLA Based Simulations," Proceedings of the Fall 2005 Simulation Interoperability Workshop and Conference, Orlando, Florida, September 2005.

Author Biographies

ROBERT G. PHILLIPS is a chief software engineer and project manager at L-3 Communications, supporting the Simulations and Robotics Division of the Engineering Directorate at Johnson Space Center. He is the lead designer and developer on the HIIA Transfer Vehicle (HTV) Flight Controller Trainer (FCT) simulation, works on the Distributed Space Exploration Simulation (DSES), and is currently helping design the HTV Space Station Training Facility (SSTF) simulation. He has developed training and flight-related software at JSC for over 18 years. He has a Masters of Computer Science from Rice University.

DAN E. DEXTER is a chief systems engineer at L-3 Communications, supporting the Simulations and Robotics Division of the Engineering Directorate at Johnson Space Center. He is the lead designer and developer of the Trick High Level Architecture (TrickHLA) simulation model and is currently working on the Distributed Space Exploration Simulation (DSES). He has developed nonlinear signal and image processing, distributed supercomputing, and flight-related software at JSC for more than 12 years. He received a B.S. degree in Electrical Engineering from the University of Kansas.

DAVID A. HASAN works for L-3 Communications, supporting the Simulations and Robotics Division of the Engineering Directorate at Johnson Space Center. He has 14 years of experience developing software, including distributed expert systems, mission control center middleware and applications, embedded GPS-based autonomous navigation systems, distributed computing systems, automated computer fault detection, and metering and billing systems for grid computing. He received B.S. degrees in Aeronautical and Astronautical Engineering and Political Science from the University of Illinois and a M.S. in Aerospace Engineering from the University of Texas.

EDWIN Z. CRUES, PH.D. has supported the Automation, Robotics and Simulation Division at NASA Johnson Space Center for the past 14 years. Since 2004, he has been a member of the Simulation and Graphics Branch where he leads the research and development of distributed simulation technologies. In this capacity, he leads the development of the HTV Flight Controller Trainer (FCT) and the NASA Distributed Space Exploration Simulation (DSES). The DSES work is in support of the Modeling and Simulation and Data Architectures (MS&DA) group for the Constellation program. Dr. Crues also supports dynamics model development for the Trick Simulation Environment and the Common Model set.

Appendix: Code Examples

Example 1: Create the Federation

```
// Try to create the federation
RTIambassadorFactory * rtiAmbFactory = new RTIambassadorFactory();
rtiAmbassador = rtiAmbFactory->createRTIAmbassador( … );
try {
   rtiAmbassador->createFederationExecution( … );
catch ( RTI::FederationExecutionAlreadyExists &e ) {
   // This is ok. Some other federate must have already created the federation.
}

// Join the federation. One of the arguments is the federate ambassador.
// If this call succeeds, then the program has become a "federate".
federate_id = rtiAmbassador->joinFederationExecution( … );
```

Example 2: Publish/Subscribe Object Attributes

```
// This is the object whose attributes we are interested in.
object_id = rtiAmbassador->getObjectClassHandle( object_name );

// These are the attributes of that object that we are interested in.
attribute_id_1 = rtiAmbassador->getAttributeHandle( object_id, attribute_name_1 );
attribute_id_2 = rtiAmbassador->getAttributeHandle( object_id, attribute_name_2 );
…
attribute_id_n = rtiAmbassador->getAttributeHandle( object_id, attribute_name_n );

// Put the attribute ids into a map. This is a collection of the attributes we
// are interested in. It does not need to include all the attributes of the object.
attributes_map = …;

// To publish the attributes in the map, we do this.
rtiAmbassador->publishObjectClassAttributes( object_id, attributes_map);

// To subscribe to the attributes in the map, we do this.
rtiAmbassador->subscribeObjectClassAttributes( object_id, attributes_map, true );
```

Example 3: Publish/Subscribe Interactions

```
// This is the interaction we are interested in.
interaction_id = rtiAmbassador->getInteractionClassHandle( interaction_name );

// To publish an interaction, we do this.
rtiAmbassador->publishInteractionClass( interaction_id );

// To subscribe to an interaction, we do this.
rtiAmbassador->subscribeInteractionClass( interaction_id );
```

Example 4: Reserve Object Instance Names

```
// Ask the RTI to reserve an object instance name.
wstring instance_name = L"MyFederateName_MyObjectName";
rtiAmbassador->reserveObjectInstanceName( instance_name );
```

Example 5: Name Reservation Callbacks

```
// This callback is invoked by the RTI when a name reservation request succeeds.
void MyFedAmbassador::objectInstanceNameReservationSucceeded(
   wstring const & theObjectInstanceName )
throw ( RTI::UnknownName, RTI::FederateInternalError )
{
   // Record the fact that the reservation of this name succeeded.
   setNameReserved( theObjectInstanceName, true );
}

// This callback is invoked by the RTI when a reservation fails.
void MyFedAmbassador::objectInstanceNameReservationFailed(
   wstring const & theObjectInstanceName )
throw ( RTI::UnknownName, RTI::FederateInternalError )
{
   setNameReserved( theObjectInstanceName, false );
   // Handle the fact that the reservation failed. This is generally fatal.
   error( … );
}
```

Example 6: Register Object Instances

```
// Wait for the RTI to confirm that the name was successfully registered.
while( ! isNameReserved( instance_name ) ) {
  usleep( 100 );
}

// Register an object instance using that name.
instance_id1 = rtiAmbassador->registerObjectInstance( object_type_id, instance_name );
```

Example 7: Object Instance Discovery Callback

```
// This callback is invoked by the RTI to notify the federate that a new object instance
// has been discovered, i.e., some other federate has registered it.
void MyFedAmbassador::discoverObjectInstance(
   ObjectInstanceHandle const & theObject,
   ObjectClassHandle const & theObjectClass,
   wstring const & theObjectInstanceName)
throw ( RTI::CouldNotDiscover,
        RTI::ObjectClassNotKnown,
        RTI::FederateInternalError )
{
   // Record the fact that this object instance has been discovered.
   saveInstance( theObject , theObjectClass, theObjectInstanceName );
}
```

Example 8: Register Synchronization Points

```
// The "initialize" synchronization point is used to mark when all federates are
// joined and ready to initialize data. Here is how we register it.
rtiAmbassador->registerFederationSynchronizationPoint( L"initialize", … );

// The "startup" synchronization point is used to determine when
// to start the simulation. Here is how we register it.
rtiAmbassador->registerFederationSynchronizationPoint( L"startup", … );
```

Example 9: Synchronization Point Registration Callbacks

```
// This callback is invoked by the RTI whenever sync point registration succeeds.
void MyFedAmbassador::synchronizationPointRegistrationSucceeded(
   wstring const & sp_label )
throw ( RTI::FederateInternalError )
{
   // We only have special logic for the "initialize" sync point.
   if ( sp_label.compare(sp_label,L"initialize") == 0) {
      set_master( true );
      set_initialize_sp_exists( true );
   } else if( sp_label.compare(sp_label,L"startup") == 0) {
      set_master( false );
      set_startup_sp_exists( true );
   }
}

// This callback is invoked by the RTI whenever sync point reservation fails.
// Sometimes this is because another federate has already reserved it.
void MyFedAmbassdor::synchronizationPointRegistrationFailed(
   wstring const & sp_label,
   SynchronizationFailureReason reason )
throw ( RTI::FederateInternalError )
{
   bool because_nonunique =
          ( reason == SynchronizationFailureReason::synchronizationPointLabelNotUnique() );

   if ( because_nonunique ) {
      if ( sp_label.compare(sp_label,L"initialize") == 0 ) {
         // Someone else registered the "initialize" sync point.
         // That means we are NOT the "master federate". But that also means that
         // the "initialize" sync point does indeed exist.
         set_master( false );
         set_initialize_sp_exists( true );
      } else if ( sp_label.compare(sp_label,L"startup") == 0 ) {
         set_startup_sp_exists( true );
      }
   } else {
      error( … );
   }
}
```

Example 10: Wait for Synchronization Point Registration

```
// Wait for both sync points to be successfully registered. These conditions should
// eventually be set to true when the RTI invokes the sync point registration callbacks
// in the federation ambassador. (See Code Example 9.)
while( ! fedAmbassador.initialize_sp_exists() && !fedAmbassador.startup_sp_exists() ) {
   usleep( 100 );
}
```

Example 11: Wait for All Federates to Join

```
// The "initialize" sync point has been registered already. (See Code Example 8.)

if( master() ) {
   // This federate is indeed the "master federate". Therefore it needs to determine
   // determine which federates have joined and to wait until they all have.
   …subscribe to joined federates…  // (See Code Example 10A.)
   …compare joined to expected federates… // (See Code Example 10B.)
} else {
   // This federate is NOT the "master federate". There is nothing else to do.
}

// All federates proceed to the "initialize" sync (described elsewhere). The
// non-masters will end up waiting for the master, at which point they can all proceed.
// (See Code Example 15.)
```

Example 12: Subscribe to Joined Federates

```
// Get the ID for the MOM federate object.
MOM_federate_object_id =
   rtiAmbassador->getObjectClassHandle( L"HLAobjectRoot.HLAmanager.HLAfederate" );
// Get the ID for the attribute we're interested in.
MOM_federate_attribute_id =
   rtiAmbassador->getAttributeHandle( MOM_federate_object_id, L"HLAfederateType" );
… Create a one-entry map, attrs, containing MOM_federate_attribute_id.
// Subscribe to the attribute.
rtiAmbassador->subscribeObjectClassAttributes( MOM_federate_object_id, attrs, true );

// Force the RTI to send an immediate data update for the subscribed to object.
// This is sometimes necessary to force the RTI to do an immediate data update.
try {
   rtiAmbassador->requestAttributeValueUpdate( MOM_federate_object_id, attrs, … );
} catch( RTI::exception & e ) {
   error( … );
}
```

Example 13: Object Discovery and Reflection Callbacks

```
// This callback is invoked by the RTI to notify the federate that a new object instance
// has been discovered, i.e., some other federate has registered it.
void MyFedAmbassador::discoverObjectInstance(
   ObjectInstanceHandle const & theObject,
   ObjectClassHandle const & theObjectClass,
   wstring const & theObjectInstanceName)
throw ( RTI::CouldNotDiscover,
        RTI::ObjectClassNotKnown,
        RTI::FederateInternalError )
{
   // Is this the MOM federate object to which we subscribed?
   int isMOMFederateObject =
      …;  // Compare theObjectClass to MOM_federate_object_id from Code Example 12.

   if ( isMOMFederateObject ) {
      … Save theObject in MOM_federate_instance_id (to recognize reflected attributes).
      … Create a joined-federates data structure (initially empty).
   } else {
      // handle other object instances
      …
   }
}

// This callback is invoked by the RTI to notify the some attribute values of an object
// to which we subscribed have been reflected.
void MyFedAmbassador::reflectAttributeValues(
   ObjectInstanceHandle const & theObject,
   std::auto_ptr< AttributeHandleValueMap > theAttributeValues,
   UserSuppliedTag const & theUserSuppliedTag,
   OrderType const & sentOrder,
   TransportationType const & theType )
throw ( RTI::ObjectInstanceNotKnown,
```

```
        RTI::AttributeNotRecognized,
        RTI::AttributeNotSubscribed,
        RTI::FederateInternalError )
{
   if ( …theObject is the same as MOM_federate_instance_id?… ) {
      // This callback API designed for the general case where federates subscribe to
      // many attributes, which is why the second argument is a map. In our case,
      // we've only subscribed to one attribute, so we expect the map to contain
      // a single value. Take it out of the map.
      AttributeHandleValueMap::iterator attribute_iterator =
         theAttributeValues.find( MOM_federate_attribute_id );
      AttributeValue attribute_value = attribute_iterator->second;

      // This value is a bunch of raw bytes. Get the bytes.
      int num_bytes = attribute_value.size();
      char* data = (char*) attribute_value.data();

      // The first four bytes represent the number of two-byte characters that
      // are the string. For example, a federate name of "CEV" would have the
      // following ASCII decimal values in the array.
      //
      //  0 0 0 3 0 67 0 69 0 86
      //  ---+---    |    |    |
      //  len = 3    C    E    V
      //
      // Assuming ASCII names, then we grab every other byte starting at the
      // sixth byte. Together they form the name of the joining federate.

      wstring joining_federate_name( L"" );
      for( int i = 5; i < num_bytes; i += 2 ) {
         joining_federate_name.append( data+i, data+i+1 );
      }

      … Add joining_federate_name to the joined_federates data structure.
   } else {
      // handle other attributes
      …
   }
}
```

Example 14: Waiting Until All Federates Have Joined

```
// Wait until all the expected federates have joined.
while ( ! all_federates_joined ) {
   if( …All entries in expected_federates are present in joined_federates?… ) {
      break; // They've all joined.
   } else {
      // Sleep a while. Maybe a new federate will join. If so, the reflect callback
      // will update joined_federates with the name of the new federate.
      usleep( 100 );
   }
}
// If we reach this point, then all the expected federates have joined,
// so we're no longer interested in the MOM federate object.
rtiAmbassador->unsubscribeObjectClass( MOM_federate_object_id );
```

Example 15: Achieve "initialize" Synchronization Point

```
// Try to achieve the "initialize" sync point. The achieve won't be successful until
// all the federates have tried this.
rtiAmbassador->synchronizationPointAchieved( L"initialize" );

// Wait for all federates to get to this point. When they do, the RTI will have
// invoked the federationSynchronized callback, and this will have set the flag
// all_federates_waiting_at_initialize_sync_point. Assuming this flag is initialized
// to false, this loop will hang until all federates are ready to proceed.
while ( … all_federates_waiting_at_initialize is false ) {
   usleep( 100 );
}
```

Example 16: Synchronization Point Achieved Callback

```
// This callback is invoked by the RTI when a sync point has been achieved.
void MyFedAmbassador::federationSynchronized( wstring const & label)
throw ( RTI::FederateInternalError )
{
   if ( label.compare( L"initialize" ) == 0 ) {
      // This means all the federates (including the master-federate) have arrived.
      // Set a flag to indicate that fact.
      … set all_federates_waiting_at_initialize to true
   }
}
```

Example 17: Federate Update with Initial Object Attribute Values

```
rtiAmbassador->enableAsynchronousDelivery();

// Now update the attributes of any relevant attributes with initial values
rtiAmbassador->updateAttributeValues( instance_1, attributeValues_1, … );
rtiAmbassador->updateAttributeValues( instance_2, attributeValues_2, … );
…
rtiAmbassador->updateAttributeValues( instance_n, attributeValues_n, … );
```

Example 18: Federate Waits for All Initial Data

```
// Don't go any further until all instances have been properly initialized.
// See Code Example 16 to where this flag eventually gets set.
while ( ! all_instances_initialized ) {
   usleep( 100 ); // or some method of non-blocking wait
}
```

Example 19: Initial Attribute Value Reflection Callback

```
// This callback is invoked by the RTI whenever new values are reflected. This
// example demonstrates how a flag indicating successful initialization may be set
// after all the initial reflections have been carried out.
void MyFedAmbassador::reflectAttributeValues(
   ObjectInstanceHandle const        & theObject,
   auto_ptr< AttributeHandleValueMap > theAttributeValues,
   UserSuppliedTag const             & theUserSuppliedTag,
   OrderType const                   & sentOrder,
   TransportationType const          & theType )
throw ( RTI::ObjectInstanceNotKnown,
        RTI::AttributeNotRecognized,
        RTI::AttributeNotSubscribed,
        RTI::InvalidLogicalTime,
        RTI::FederateInternalError )
{
   if ( theObject == instance_a->get_instance_id() ) {
      instance_a->reflect_data( *theAttributeValues );
      if ( ! instance_a->initialized() )
         instance_a->set_initialized( true );
   }
   else if ( theObject == instance_b->get_instance_id() ) {
      instance_b->reflect_data( *theAttributeValues );
      if ( ! instance_b->initialized() )
         instance_b->set_initialized( true );
   }
   else if ( theObject == instance_c->get_instance_id() ) {
      instance_c->reflect_data( *theAttributeValues );
      if ( ! instance_c->initialized() )
         instance_c->set_initialized( true );
   …etc…
   }
   check_all_instances_initialized();
}
```

Example 20: Initialize Time Management

```
rtiAmbassador->enableTimeConstrained();
rtiAmbassador->enableTimeRegulation( lookahead_interval );
```

Example 21: Achieve `startup` Synchronization Point

```
rtiAmbassador->synchronizationPointAchieved( L”startup” );
while ( ! startup_sp_synchronized ) {
   usleep( 100 ); // or some method of non-blocking wait
}
```

Example 22: Ready to Start Simulation

```
void MyFedAmbassador::federationSynchronized(
   wstring const & label)
throw ( RTI::FederateInternalError )
{
   ...
   if ( label.compare( L"startup" ) == 0 ) {
      federate->set_startup_sp_synchronized( true );
   }
}
```