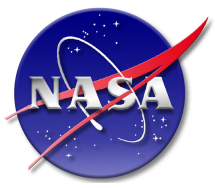
Distributed Space Exploration Simulation

Multiphase Initialization Design

Dan E. Dexter
L-3 Communications Inc.

Written for the
Simulation and Graphics Branch (ER7)
Automation, Robotics and Simulation Division (ER)
Engineering Directorate
Lyndon B. Johnson Space Center

Date: June 7, 2007
Revision: *DRAFT 2*



National Aeronautics and Space Administration

Acknowledgments

The Distributed Space Exploration Simulation (DSES) multiphase initialization design is the result of collaboration among the following individuals Mike Blum, Edwin Z. Crues, Dan Dexter, Jim Gibson, David Hasan, Joe Hawkins, Robert Phillips, Mark Ricci, and Paul Sugden.

Table Of Contents

Table of Figures

1 Introduction

This document describes the design of the Distributed Space Exploration Simulation (DSES) federate multiphase initialization process. The main goal of multiphase initialization is to allow for data interdependencies during the federate initialization process.

DSES uses the High Level Architecture (HLA) IEEE 1516 [1] to provide the communication and coordination between the distributed parts of the simulation. They are implemented using the Runtime Infrastructure (RTI) from Pitch Technologies AB. This document assumes a basic understanding of IEEE 1516 HLA, and C++ programming. In addition, there are several subtle points in working with IEEE 1516 and the Pitch RTI that need to be understood, which are covered in Appendix A

2 Requirements

DSES multiphase initialization has the following requirements:

1. DSES multiphase initialization shall support initialization data interdependencies among federates.
2. The data and optional synchronization point associated with each initialization phase shall be predetermined and known by all federates ahead of time.

3 Overview

The DSES multiphase initialization consists of three high-level stages, namely federation creation, simulation configuration, and federate initialization. Where the simulation configuration stage also depends on whether the federate is the "master" or not. Figure 1 below shows the relationship of these stages, which is a high-level representation of the DSES multiphase initialization process flowchart shown in Appendix B.

The DSES multiphase initialization process will be documented using the following outline:

1. Federation Creation
2. Simulation Configuration
   a. Master Federate Specific Configuration
   b. Non-Master Federate Specific Configuration
3. Federate Initialization

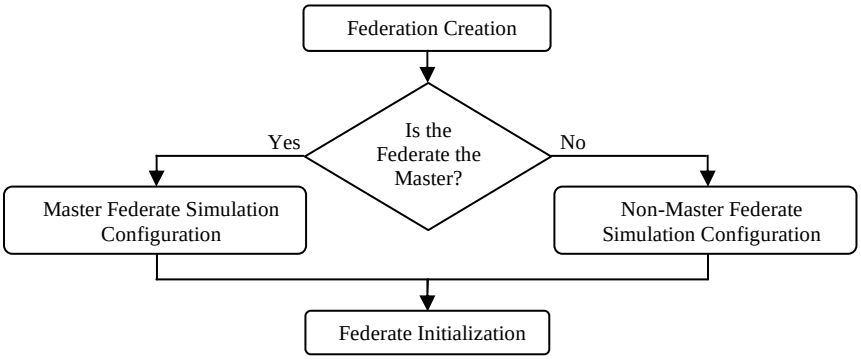


Figure 1 DSES Multiphase Initialization High-Level Representation

4 Federation Creation

The following steps comprise the "Federation Creation" stage of the multiphase initialization process shown in Figure 1, which corresponds to the top section of the multiphase initialization flowchart shown in Appendix B.

4.1 Create the Federation

This section involves creating and initializing the federation.

```
rtiAmbassador = RTI::createFederateAmbassador( args ) // host, port
```

The RTI Ambassador is the object through which the Federate requests services ***from*** the RTI. The Federate Ambassador is the object through which the RTI sends communications ***to*** the Federate.

```
rtiAmbassador->createFederationExecution( federation_name, FDD_file_name );
```

Standard HLA practice is that all federates should attempt to create the federation. The first one should succeed, and the other attempts should generate a "Federation already exists" exception. This should be accepted as equivalent to a successful creation.

4.2 Join the Federation

Attempts to join the federation are performed in a loop with multiple tries and a timeout failure, because sometimes the attempt to join failed. More recent versions of the Pitch RTI have not been checked to see if this problem still occurs.

```
rtiAmbassador->joinFederationExecution( federate_name, federation_name,
      federate_ambassador, time_factory, interval_factory );
```

4.3 Enable Asynchronous Delivery

Asynchronous delivery will need to be enabled so that Receive Order (RO) data reflections and certain other calls from the RTI arrive when the federate is not in time advancement mode. This is critical for receipt of data when the federate is paused or during the multiphase initialization process.

```
rtiAmbassador->enableAsynchronousDelivery();
```

4.4 Reserve "SimConfig" Object Instance Name

The "SimConfig" name will be reserved for the *Simulation Configuration* object instance. For the reserved name to be valid, it must be (1) unique in the Federation, and (2) *not* start with "HLA". Since the "SimConfig" object instance name can only be registered once for the federation, the success or failure to reserve the name will be used to determine the "master" federate.

```
wstring instance_name = L"SimConfig";

rtiAmbassador->reserveObjectInstanceName( instance_name );
```

4.5 Wait for "SimConfig" Object Instance Name Reservation Callback

The federate must wait for the name reservation success or failure to be acknowledged with an asynchronous callback to the federate ambassador. If the "SimConfig" name registration succeeded then the federate is the "master" federate. If the "SimConfig" name registration failed then it is because the name is already reserved by another federate and the federate is not the master. For all other name registration failures the federate is probably in an inconsistent state because of a naming conflict with another federate and should exit gracefully.

```
void MyFedAmbassador::objectInstanceNameReservationSucceeded(instance_name) {
   if ( instance_name.compare( L"SimConfig" ) == 0 ) {
      federate->set_master( true );
   }
   federate->set_name_reserved( instance_name );
}

void MyFedAmbassador::objectInstanceNameReservationFailed( instance_name ) {
   if ( instance_name.compare( L"SimConfig" ) == 0 ) {
      federate->set_master( false );
      federate->set_name_reserved( instance_name );
   } else {
      fatalError(L"Failed to register object instance name", instance_name);
   }
}
```

Meanwhile the federate uses a spin-lock to wait for the "SimConfig" name reservation success/failed callback from the RTI.

```
// Waiting for the set_name_reserved( L"SimConfig" ) callback
while ( ! simconfig_name_reserved ) {
   usleep( 100 );
}
```

5 Master Federate Simulation Configuration

The following steps comprise the master federate "Simulation Configuration" stage of the multiphase initialization process shown in Figure 1, which corresponds to the left branch of the multiphase initialization flowchart shown in Appendix B.

5.1 Publish and Subscribe

The master federate must **publish** the *Simulation Configuration* object as well as publish and subscribe to all relevant HLA objects and interactions. Publishing means "announcing to the RTI a federate's intention to create and update object instances and interactions," not actually sending data.

```
object_id    = rtiAmbassador->getObjectClassHandle( object_name );
attribute_id = rtiAmbassador->getAttributeHandle( object_id, attribute_name );
interaction_id  = rtiAmbassador->getInteractionClassHandle( interaction_name );
parameter_id = rtiAmbassador->getParameterhandle(interaction_id, parameter_name);
```

In order to work with the RTI, the federate needs unique identifiers for all objects, object attributes, interactions, and interaction parameters. The above statements are used to gain these identifiers.

```
rtiAmbassador->publishObjectClassAttributes( object_id, map );
rtiAmbassador->subscribeObjectClassAttributes( object_id, map, true );
rtiAmbassador->publishInteractionClass( interaction_id );
rtiAmbassador->subscribeInteractionClass( interaction_id );
```

The calls for publishing and subscribing objects require maps that contain the identifiers of all of the object attributes that will be published or subscribed to. Since interactions don't persist and all interaction parameters are sent with any interaction creation/update, such maps are not needed for interaction publishing and subscribing.

5.2 Reserve Object Instance Names

The master federate next reserves all the relevant object instance names, **excluding** the previously registered "SimConfig" name, following the procedures outlined in section 4.4 above. The names to be used in a federation should be agreed upon in advance.

5.3 Wait for All Object Instance Name Reservations

The master federate next waits for all the object instance name reservation callbacks following the procedure outlined in section 4.5 above, but with a reserved status flag specific to each object instance name.

5.4 Register Object Instances

Now that the master federate has published all objects, it needs to create instances of the objects to update **including** the *Simulation Configuration* object. This is not necessary for object instances that are not present at startup, and is not even necessary for instances that are not used during initialization. But, since it is required for instances used during initialization, it is probably best to do all of the instance creation at this point.

It is possible, even probable, that multiple federates in a given simulation will create object instances for the same object (remember, in HLA terminology, an "object" is a like a Java/C++ type or class, and an instance is a specific instantiation of that type/class). This means that a given federate that subscribes to that object could discover multiple object instances, and it will need to be able to distinguish between them.

If multiple instances of a given object will be registered in a federation (either by a single federate or by multiple federates), then the discovering federates will need some mechanism to distinguish between the separate instances. HLA provides a mechanism for precisely this circumstance by allowing federates to reserve unique object instance names.

The federate can register an instance of the object using a reserved instance name with the following call to the RTI.

```
// The array object[] contains information about the objects used by the
// federate. For this example it contains the object instance name and
// a flag indicating if the object has been registered with the RTI.
wstring instance_name = object[n].get_instance_name();

instance_id = rtiAmbassador->registerObjectInstance(
      object_type_id, instance_name ); // registering with a unique name

instance_id = rtiAmbassador->registerObjectInstance(
      object_type_id ); // registering without a unique name
```

The federate should also keep track of which object instances it has registered so that the check for registered object instances that occurs in section 5.5 below will pass for this object.

```
object[n].set_registered( true );
```

When a publishing federate makes the above call to the RTI, the RTI will respond with a subsequent call to all subscribing federates through the Federate Ambassador as follows:

```
void MyFedAmbassador::discoverObjectInstance(instance_id, object_id, instance_name)
{
   // Save the instance ID and mark the object as registered.
   saveInstance( object_id, instance_id, instance_name );
}
```

The discovered object instance with given instance name should likewise be marked as being registered so that the check for registered object instances in the next section will pass for this object. If the registering federate reserved and used a unique object instance name when registering the object instance, then the `instance_name` parameter will contain that reserved name. Otherwise, the Pitch RTI will assign a generated unique name that starts with "HLA".

5.5 Wait for All Objects to be Registered

Next the federate waits for all the object instances it will be using during the course of the simulation to be registered with the RTI. This step ensures that both the object instances the federate will be updating as well as the object instances it will receive reflected attribute changes for exist before initialization continues. Any mistake in the list of object instances or object instance names the federate expects to use can be detected in this step. The federate will wait indefinitely for all the object instances to be registered as a result of the RTI callback to the `MyFedAmbassador::discoverObjectInstance()` function, which in turn should mark the object for the specific instance as registered.

```
bool any_unregistered_obj;
do {
   any_unregistered_obj = false;

   // All registered objects should have a non-zero object instance handle.
   for ( int n = 0; n < obj_count; n++ ) {

      if ( ! object[n].is_registered() ) {
         any_unregistered_obj = true;

         // Optionally, display a message about the unregistered object.
```

```
      }
   }

   usleep( 100 );

} while ( any_unregistered_obj );
```

5.6 Wait for All "Required" Federates to Join

Now that the master federate has successfully joined the federation, has successfully published and subscribed to all necessary objects and interactions, and has created all object instances (especially the *Simulation Configuration* object and those needed for exchange of initialization data), the next step is to ensure that all other required federates are also joined to the federation. All federates could perform the necessary check, but to speed up the initialization process, only the master federate performs the check.

The master federate uses the Management Object Model (MOM) interface to check for the existence of all required federates. It is assumed that each federate has access to a list of required federates, and that the required federates have unique names.

```
// Get the ID for the MOM federate object
federate_type_id = rtiAmbassador->getObjectClassHandle(
      L"HLAobjectRoot.HLAmanager.HLAfederate" );

// Get the ID for the name attribute of the federate object
federate_name_type_id = rtiAmbassador->getAttributeHandle(
      federate_type_id, L"HLAfederateType" );

// Create a map with the desired attribute
fedAttributes.insert( federate_name_type_id );

// Subscribe to the federate object and the name attribute
rtiAmbassador->subscribeObjectClassAttributes( federate_type_id,
      fedAttributes, true );

// This code forces the RTI to send an immediate data update for the
// subscribed to object. This is sometimes necessary with the Pitch RTI
// to force an immediate data update.
try {
   attributes.insert( federate_name_type_id );
   rtiAmbassador->requestAttributeValueUpdate( federate_type_id, attributes,
         UserSuppliedTag( 0, 0 ) );
} catch( RTI::exception & e ) {
   cerr << "Update request failed!" << endl;
   return;
}

all_federates_joined = false;
while ( ! all_federates_joined ) {

   // This loop causes the main federate thread to periodically check
   // until it has determined that all required federates are joined to
   // the federation.
   usleep( 100 );
   all_federates_joined = true;

   bool found_it;
   map< ObjectInstanceHandle, wstring >::iterator fed_iter;

   for ( int i = 0; i < timeout_count; i++ ) {
      found_it = false;
      for ( fed_iter = federate_instances.begin();
```

```
            fed_iter != federate_instances.end(); fed_iter++ ) {
         if ( fed_iter->second.compare( requiredFederates[i] ) == 0 ) {
            found_it = true;
         }
      }
      if ( ! found_it ) {
         all_federates_joined = false;
      }
   }
}
rtiAmbassador->unsubscribeObjectClass( federate_type_id );
```

As a result of subscribing to the Federate attributes above we will get asynchronous calls to `MyFedAmbassador::discoverObjectInstance()` to provide the object instance handles for other MOM federate objects, and `MyFedAmbassador::reflectAttributeValues()` to reflect the federate name values.

A Standard Template Library (STL) map (`map< ObjectInstanceHandle, wstring >`) of federates will be maintained to keep track of the federate object instance handles and associated names. As federates are discovered they are added to the map.

```
void MyFedAmbassador::discoverObjectInstance (
   ObjectInstanceHandle const & theObject,
   ObjectClassHandle const & theObjectClass,
   wstring const & theObjectInstanceName)
throw (
   RTI::CouldNotDiscover,
   RTI::ObjectClassNotKnown,
   RTI::FederateInternalError )
{
   if ( myfederate && (theObjectClass == myfederate->federate_type_id) ) {
      myfederate->add_federate_instance_id( theObject );
   } else {
      // handle discovery of other object instances
   }
}
```

The `add_federate_instance_id()` function is used to maintain the list of federates.

```
void MyFederate::add_federate_instance_id(
   ObjectInstanceHandle instance_id )
{
   // Default to an empty string for the federate name. We will store the
   // actual name when it is reflected.
   federate_instances[ instance_id ] = L"";
}
```

The federate names are reflected back to the federate ambassador. Once it is determined the attribute values are for a federate by checking against the object instance handle, the federate name is set.

```
void MyFedAmbassador::reflectAttributeValues (
   ObjectInstanceHandle const & theObject,
   std::auto_ptr< AttributeHandleValueMap > theAttributeValues,
   UserSuppliedTag const & theUserSuppliedTag,
   OrderType const & sentOrder,
   TransportationType const & theType )
throw (
   ObjectInstanceNotKnown,
   AttributeNotRecognized,
```

```
   AttributeNotSubscribed,
   FederateInternalError)
{
   if ( myfederate && myfederate->is_federate_instance_id( theObject ) ) {
      myfederate->set_federate_instance_name(theObject, theAttributeValues);
   } else {
      // handle the other object attributes
   }
}
```

The federate name is found in the map of attribute handle values by using the attribute handle `federate_name_type_id`, which corresponds to the `HLAfederateType` attribute that was subscribed to. The federate name, in the `HLAunicodeString` format, is decoded using the following code which uses the technique described in Appendix A, section A.3 below.

```
void MyFederate::set_federate_instance_name(
   ObjectInstanceHandle                id,
   auto_ptr< AttributeHandleValueMap > values )
{
   map< ObjectInstanceHandle, wstring >::iterator federate_iter;
   AttributeHandleValueMap::iterator attr_iter;

   // Find the federate for the given ID.
   federate_iter = federate_instances.find( id );

   // Find the attribute handle value for the given attribute name type ID.
   attr_iter     = values->find( federate_name_type_id );

   if ( ( federate_iter != federate_instances.end() )
                  && ( attr_iter != values->end() ) ) {

      wstring wname(L"");
      int    size = attr_iter->second.size();
      char * data = (char *) attr_iter->second.data();

      // The first four bytes represent the number of two-byte characters
      // that are in 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
      //
      for ( int i = 5; i < size; i += 2 ) {
         wname.append( data+i, data+i+1 );
      }

      federate_instances[ id ] = wname;
   }
}
```

5.7 Register Synchronization Points

The master federate is responsible for registering the “sim_config”, “initialize”, and “startup” synchronization points as well as all the multiphase initialization synchronization points if any exist. The following calls create the synchronization points needed for the federate multiphase initialization process.

```
rtiAmbassador->registerFederationSynchronizationPoint( L"sim_config",
      UserSuppliedTag( 0, 0 ) );

rtiAmbassador->registerFederationSynchronizationPoint( L"initialize",
```

```
      UserSuppliedTag( 0, 0 ) );

rtiAmbassador->registerFederationSynchronizationPoint( L"startup",
      UserSuppliedTag( 0, 0 ) );

// Register all the Multiphase Initialization Sync-Points if we have any.
for ( int i = 0; i < num_multiphase_sp; i++ ) {
   rtiAmbassador->registerFederationSynchronizationPoint( multiphase_sp[i],
         UserSuppliedTag( 0, 0 ) );
}
```

The Federate Ambassador receives callbacks from the RTI to indicate synchronization point registration success or failure. For each successful synchronization point registration, a flag is set to indicate that the synchronization point exists. Nothing is done for the successful registration of synchronization points that are not recognized by the federate.

```
MyFedAmbassador::synchronizationPointRegistrationSucceeded(
      wstring sp_label )
{
   if ( sp_label.compare( L"sim_config" ) == 0 ) {
      federate->set_sim_config_sp_exists( true );
   }
   else if ( sp_label.compare( L"initialize" ) == 0 ) {
      federate->set_initialize_sp_exists( true );
   }
   else if ( sp_label.compare( L"startup" ) == 0 ) {
      federate->set_startup_sp_exists( true );
   }
   else if ( federate->is_multiphase_sp( sp_label ) ) {
      federate->set_multiphase_sp_exists( sp_label, true );
   }
}
```

Only the master federate is creating all the synchronization points so any registration failure should be handled as an error and gracefully exit the simulation.

```
MyFedAmbassdor::synchronizationPointRegistrationFailed(
   wstring sp_label,
   SynchronizationFailureReason reason )
{
   fatalError( L“Failed to register synchronization point”, sp_label );
}
```

5.8 Wait for Synchronization Point Registration Confirmations

Next the master federate waits for confirmation that all the synchronization points have successfully registered.

```
while ( ! all_sp_exist() ) {
   usleep( 100 );
}
```

The `all_sp_exist()` function returns true once the “sim_config”, “initialize”, “startup”, and multiphase synchronization points all exist.

```
bool MyFederate::all_sp_exist()
{
   if ( ! sim_config_sp_exists ||
```

```
        ! initialize_sp_exists ||
        ! startup_sp_exists ) {
      return false;
   }
   for ( int i = 0; i < num_multiphase_sp; i++ ) {
      if ( ! multiphase_sp_exists[i] ) {
         return false;
      }
   }
   return true;
}
```

5.9 Achieve "sim_config" Synchronization Point

At the end of the check for all federates, the master federate is guaranteed that all required federates are joined. Since the master federate will not advance to this step until then, it is guaranteed that all federates will achieve the "sim_config" synchronization point only after all required federates are joined, and all object instances required for initialization have been created.

```
rtiAmbassador->synchronizationPointAchieved( L"sim_config" );
```

5.10 Wait for "sim_config" Federation Synchronization

Meanwhile, the following code in the Federate Ambassador is called by the RTI when all federates have achieved a particular synchronization point:

```
void MyFederateAmbassador::federationSynchronized (
   wstring const & sp_label)
   throw ( RTI::FederateInternalError )
{
   if ( sp_label.compare( L"sim_config" ) == 0 ) {
      federate->set_sim_config_sp_synchronized( true );
   }
   else if ( sp_label.compare( L"initialize" ) == 0 ) {
      federate->set_initialize_sp_synchronized( true );
   }
   else if ( sp_label.compare( L"startup" ) == 0 ) {
      federate->set_startup_sp_synchronized( true );
   }
   else if (federate->is_multiphase_sp( sp_label ) ) {
      federate->set_multiphase_sp_synchronized( sp_label, true );
   }
}
```

The federate will wait in a loop until the "sim_config" synchronization point has been synchronized for the federation.

```
while ( ! sim_config_sp_synchronized ) {
   usleep( 100 );
}
```

5.11 Update *Simulation Configuration* Object Attributes

This step causes a federate to send data for the *Simulation Configuration* object instance to subscribing federates.

```
// sim_config_instance is a pointer to a user-defined class for managing
// an instance of a Simulation-Configuration object
AttributeHandleValueMap * map = sim_config_instance->get_attribute_map();
```

```
// No timestamp is used, so the data will be in Receive Order (RO).
rtiAmbassador->updateAttributeValues(
      sim_config_instance->get_id(),
      map,
      UserDefinedTag( 0, 0 ) );
```

If the *Simulation Configuration* object instance is not going to be used again, it can now be deleted.

```
rtiAmbassador->deleteObjectInstance( sim_config_instance->get_id(),
      UserDefinedTag( 0, 0 ) );
```

If no instances of the *Simulation Configuration* object are going to be published any more, the object can now be unpublished.

```
rtiAmbassador->unpublishObjectClass( sim_config_instance->get_object_id() );
```

6 Non-Master Federate Simulation Configuration

The following steps comprise the non-master federate "Simulation Configuration" stage of the multiphase initialization process shown in Figure 1, which corresponds to the right branch of the multiphase initialization flowchart shown in Appendix B.

6.1 Publish and Subscribe

The non-master federate must **subscribe** to the *Simulation Configuration* object as well as publish and subscribe to all relevant HLA objects and interactions. The procedures outlined in section 5.1 are followed with the exception that the *Simulation Configuration* object is only subscribed to and not published.

6.2 Reserve Object Instance Names

The non-master federate next reserves all the relevant object instance names, **excluding** the already registered "SimConfig" name, following the procedures outlined in section 4.4 above. The names to be used in a federation should be agreed upon in advance.

6.3 Wait for All Object Instance Name Reservations

The non-master federate next waits for all the object instance name reservation callbacks following the procedure outlined in section 4.5 above, but with a reserved status flag specific to each object instance name.

6.4 Register Object Instances

Now that the non-master federate has published all objects, it needs to create instances of the objects to update **excluding** the *Simulation Configuration* object. The procedures in section 5.4 above are followed with the exception that an instance for the *Simulation Configuration* object is not created.

6.5 Wait for All Objects to be Registered

Now that the non-master federate has created the object instances it will be updating, it needs to wait for all the object instances it will be using to be registered by following procedures in section 5.5 above. The object instances the non-master federate uses include those it will update, the *Simulation Configuration*, and those it will receive RTI reflections for from the other federates.

6.6 Wait for Announcement of Synchronization Points

The Federate Ambassador receives callbacks from the RTI announcing the existence of a synchronization point that applies to the federate. Nothing is done for announced synchronization points that are not recognized by the federate.

```
void MyFedAmbassador::announceSynchronizationPoint (
       wstring const & sp_label,
       UserSuppliedTag const & theUserSuppliedTag)
throw ( RTI::FederateInternalError )
{
   if ( sp_label.compare( L"sim_config" ) == 0 ) {
      federate->set_sim_config_sp_exists( true );
   }
   else if ( sp_label.compare( L"initialize" ) == 0 ) {
      federate->set_initialize_sp_exists( true );
   }
   else if ( sp_label.compare( L"startup" ) == 0 ) {
      federate->set_startup_sp_exists( true );
   }
   else if ( federate->is_multiphase_sp( sp_label ) ) {
      federate->set_multiphase_sp_exists( sp_label, true );
   }
}
```

Next the non-master federate waits for all the synchronization points to be announced indicating that they exist.

```
while ( ! all_sp_exist() ) {
   usleep( 100 );
}
```

6.7 Achieve “sim_config” Synchronization Point

Next the non-master federate achieves the “sim_config” synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L”sim_config” );
```

6.8 Wait for “sim_config” Federation Synchronization

The non-master federate follows the same procedures outline in section 5.10 above to wait for the “sim_config” federation synchronization.

6.9 Wait for *Simulation Configuration* Object Reflections

In this step, the non-master federate executes a wait loop until it receives an update of the *Simulation Configuration* object from the master federate.

```
while ( ! sim_config_instance->is_initialized() ) {
   usleep( 100 );
}
```

If the non-master federate doesn't care about any additional updates for instances of the *Simulation Configuration* object, it can now unsubscribe to it.

```
rtiAmbassador->unsubscribeObjectClass( sim_config_type_id );
```

In the Federate Ambassador, something like the following code will store the reflected *Simulation Configuration* and initialization values from the other federates.

```
void MyFedAmbassador::reflectAttributeValues (
   ObjectInstanceHandle const        & theObject,
   auto_ptr< AttributeHandleValueMap > theAttributeValues,
   UserSuppliedTag const             & theUserSuppliedTag,
   OrderType const                   & sentOrder,
   TransportationType const          & theType )
throw (
   ObjectInstanceNotKnown,
   AttributeNotRecognized,
   AttributeNotSubscribed,
   InvalidLogicalTime,
   FederateInternalError)
{
   if ( sim_config_instance &&
        ( theObject == sim_config_instance->get_instance_id() ) ) {
      sim_config_instance->reflect_data( *theAttributeValues );
      sim_config_instance->set_initialized( true );
   }
   else if ( instance_a && ( theObject == instance_a->get_instance_id() ) ) {
      instance_a->reflect_data( *theAttributeValues );
      instance_a->set_initialized( true );
   }
   else if ( instance_b && ( theObject == instance_b->get_instance_id() ) ) {
      instance_b->reflect_data( *theAttributeValues );
      instance_b->set_initialized( true );
   }
   else if ( instance_c && ( theObject == instance_c->get_instance_id() ) ) {
      instance_c->reflect_data( *theAttributeValues );
      instance_c->set_initialized( true );
   }
}
```

7 Federate Initialization

The following steps comprise the "Federate Initialization" stage of the multiphase initialization process shown in Figure 1, which corresponds to the last page of the multiphase initialization flowchart shown in Appendix B.

7.1 Achieve "initialize" Synchronization Point
7.2 Wait for "initialize" Federation Synchronization
7.3 Send Initialization Data for the Current Phase

7.1 Achieve “initialize” Synchronization Point

The federate achieves the “initialize” synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L”initialize” );
```

7.2 Wait for “initialize” Federation Synchronization

The federate will wait in a loop until the “initialize” synchronization point has been synchronized for the federation. See section 5.10 for the Federate Ambassador federation synchronized callback.

```
while ( ! sim_config_sp_synchronized ) {
   usleep( 100 );
}
```

7.3 Send Initialization Data for the Current Phase

The federate sends the initialization data to the subscribing federates only if it needs to for the current phase.

```
// init_instances is a pointer to an array of user-defined classes for
// managing instances of initialization objects for a specific phase
int init_instances_cnt = get_init_instances_count_for( phase );
init_instances         = get_init_instances_for( phase );

for ( int i = 0; i < init_instances_cnt; i++ ) {

   if ( init_instances[i].is_send_needed_for_phase() ) {

      AttributeHandleValueMap * map = init_instances[i].get_attribute_map();

      // No timestamp is used, so the data will be in Receive Order (RO).
      rtiAmbassador->updateAttributeValues(
            init_instances[i].get_id(),
            map,
            UserDefinedTag( 0, 0 ) );
   }
}
```

7.4 Receive Initialization Data for the Current Phase

The federate executes a wait loop until it receives an update of the initialization data from the other federates and only if it needs to for the current phase.

```
// init_instances is a pointer to an array of user-defined classes for
// managing instances of initialization objects for a specific phase
int init_instances_cnt = get_init_instances_count_for( phase );
init_instances         = get_init_instances_for( phase );

for ( int i = 0; i < init_instances_cnt; i++ ) {

   if ( init_instances[i].is_receive_needed_for_phase() ) {
```

```
      while ( ! init_instances[i].is_initialized() ) {
         usleep( 100 );
      }
   }
}
```

7.5 Process Synchronization Point for the Current Phase

Next the federate achieves the "sim_config" synchronization point and waits for federation synchronization if it needs to for the current phase.

```
// init_instances is a pointer to an array of user-defined classes for
// managing instances of initialization objects for a specific phase
int init_instances_cnt = get_init_instances_count_for( phase );
init_instances         = get_init_instances_for( phase );

for ( int i = 0; i < init_instances_cnt; i++ ) {

   if ( init_instances[i].is_sync_pt_needed_for_phase() &&
        ! init_instances[i].is_sync_pt_synchronized() ) {

      rtiAmbassador->synchronizationPointAchieved(
               init_instances[i].get_sync_pt_label() );

      while ( ! init_instances[i].is_sync_pt_synchronized() ) {
         usleep( 100 );
      }
   }
}
```

7.6 Determine if Additional Initialization Phases Exist

The federate shall keep looping through the steps of sending (section 7.3), receiving (section 7.4), and processing synchronization points (section 7.5) for the initialization data for each phase until there are no more initialization phases left. When there are no more initialization phases remaining the federate will proceed to the "setup time management" (section 7.7) step.

```
int phase = 0;
do {
   send_init_data_for_phase();

   receive_init_data_for_phase();

   process_sync_pt_for_phase();
} while ( ( ++phase ) < num_phases );
```

7.7 Setup Time Management

7.7.1 Time Frames

Time management defines how the RTI synchronizes and relates the time for various federates. 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 HLA federate must operate in several distinct time frames simultaneously:

- *Real Time* is the computer's concept of time in the physical world (i.e., wall clock time).

- *Simulation Time* is the time for which the federation is calculating simulated data.
- *RTI Time* is the time for which the federate receives data from other federates.
- *Update Time* is the earliest time for which the federate can send data to other federates.

7.7.2 Real Time

The first of these time frames is referred to as *real time*. Real time 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. This time frame is often referred to as "wall clock time".

This time frame is important when a simulation is interfacing with time critical hardware or software or has elements that provide or require human interaction.

7.7.3 Simulation Time

The second of these time frames is simulation time. This is the natural time scale for the dynamic systems being simulated. From the simulation's (and therefore a federate's) point of view, this is its "real" 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 based orbital dynamics simulations are often propagated in 0.01-second time steps (100 Hz). However, Trick based robotic simulations are often propagated at 0.001-second time steps (1000 Hz).

A simulation ***can*** only run real time if it can advance its simulation time at a rate greater than or equal to real time. A simulation ***will*** only run real time if simulation time is held to the same rate as real time.

7.7.4 RTI Time

The third of these time frames is the *RTI time*. This is the time that the RTI thinks the federate is at, and therefore the time for which the RTI sends data reflections. Since the RTI time advances at a different rate (1Hz for all federates in the DIS 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.

7.7.5 Update Time

The fourth time frame is the *update time* -- the earliest time for which the federate is allowed to publish. This time 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.

7.7.6 Time Management

The following calls initialize the time management for a simulation that follows the DSES standard:

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

The default mode of operation is to have all federates be both time regulating and time constrained. The time regulation uses a lookahead interval equal to the rate of data sending, e.g., 4 Hz sending with a 250-millisecond lookahead. Keeping these two values the same eliminates the need for federates to have to queue up incoming value updates.

7.8 Achieve "startup" Synchronization Point

Next the federate achieves the "startup" synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L"startup" );
```

7.9 Wait for "startup" Federation Synchronization

The federate will wait in a loop until the "startup" synchronization point has been synchronized for the federation. See section 5.10 for the Federate Ambassador federation synchronized callback.

```
while ( ! startup_sp_synchronized ) {
   usleep( 100 );
}
```

Once the federation has been synchronized on the "startup" synchronization point the multiphase initialization is complete and the simulation can begin running.

8 References

[1] 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.* Technical Report IEEE-1516.1-2000, The Institute of Electrical and Electronics Engineers, 2 Park Avenue, New York, NY 10016-5997, September 2000.

[2] Simulation Interoperability Standards Committee (SISC) of the IEEE Computer Society. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification.* Technical Report IEEE-1516.2-2000, The Institute of Electrical and Electronics Engineers, 2 Park Avenue, New York, NY 10016-5997, September 2000.

A Subtle Points in Working with IEEE 1516

There are several subtle points in working with IEEE 1516 and the Pitch RTI that need to be understood.

A.1 Calling the RTI and Threads

All calls ***to*** the RTI are done through the RTI Ambassador. All calls ***from*** the RTI come to the Federate Ambassador. The calls to the Federate Ambassador are on threads that are created and managed by the RTI, and are independent of the main execution thread. It is *not allowed* to make a call back to the RTI through the RTI Ambassador on one of these threads. Doing so will result in undefined behavior. If the federate gets a call to its Federate Ambassador that should result in a call through the RTI Ambassador back to the RTI, the federate must do one of the following:

a. Create a brand new thread that performs the call to the RTI Ambassador.

b. Set a flag that is monitored by the main execution thread, and will cause the main thread to perform the call to the RTI Ambassador.

A.2 Wide Strings

The RTI is written in Java and uses Unicode wide strings for all test representation. If a federate is written in C++ and uses the C++ RTI bindings, it will almost certainly have to convert between wide strings and normal C++ "char *" strings. The following code is a quick and efficient method for doing this without having to explicitly allocate and manage memory. (This and all subsequent code examples are written in C++ code fragments and use the C++ RTI bindings. Equivalent functionality in Java or Ada should be fairly straightforward to implement.)

char * to wstring:

```
// c_str is of type "char *"
string str( c_str );
wstring wstr;
wstr.assign( str.begin(), str.end() );
```

wstring to char *:

```
string str;
str.assign( wstr.begin(), wstr.end() );
strcpy( c_str, str.c_str() );
```

A.3 HLAunicodeString

The `HLAunicodeString` type is used for the strings in the Management Object Model (MOM) interface, and can be used for federation-specific data.

The `HLAunicodeString` is defined in IEEE Standard 1516.2-2000, section 4.12.6 as being encoded as `HLAvariableArray` and contains elements that are `HLAunicodeChar`. The `HLAunicodeChar` is represented as `HLAoctetPairBE` of a Unicode UTF-16 character in Big Endian [2].

The `HLAvariableArray` is defined in IEEE Standard 1516.2-2000, section 4.12.9.4 as an encoding for arrays of variable length and consists of the number of encoded elements as `HLAinteger32BE` followed by the encoding of each element in sequence [2].

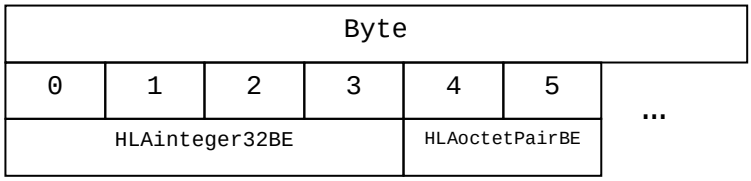


Figure 2 HLAunicodeString Format

To make things a little clearer, here is an example of a federate name of "CEV" in the `HLAunicodeString` format.

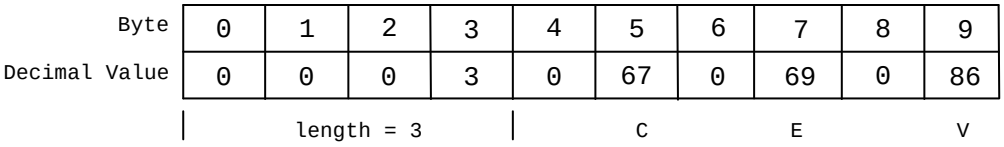


Figure 3 "CEV" in the HLAunicodeString Format

The following code shows a quick way to extract a string from the `HLAunicodeString` format held within an `AttributeHandleValue`. Characters for the string are extracted starting at index 5 which will skip over the size encoding and start with the first decodable character. Also, every other character is used for the string to decode the UTF-16 values.

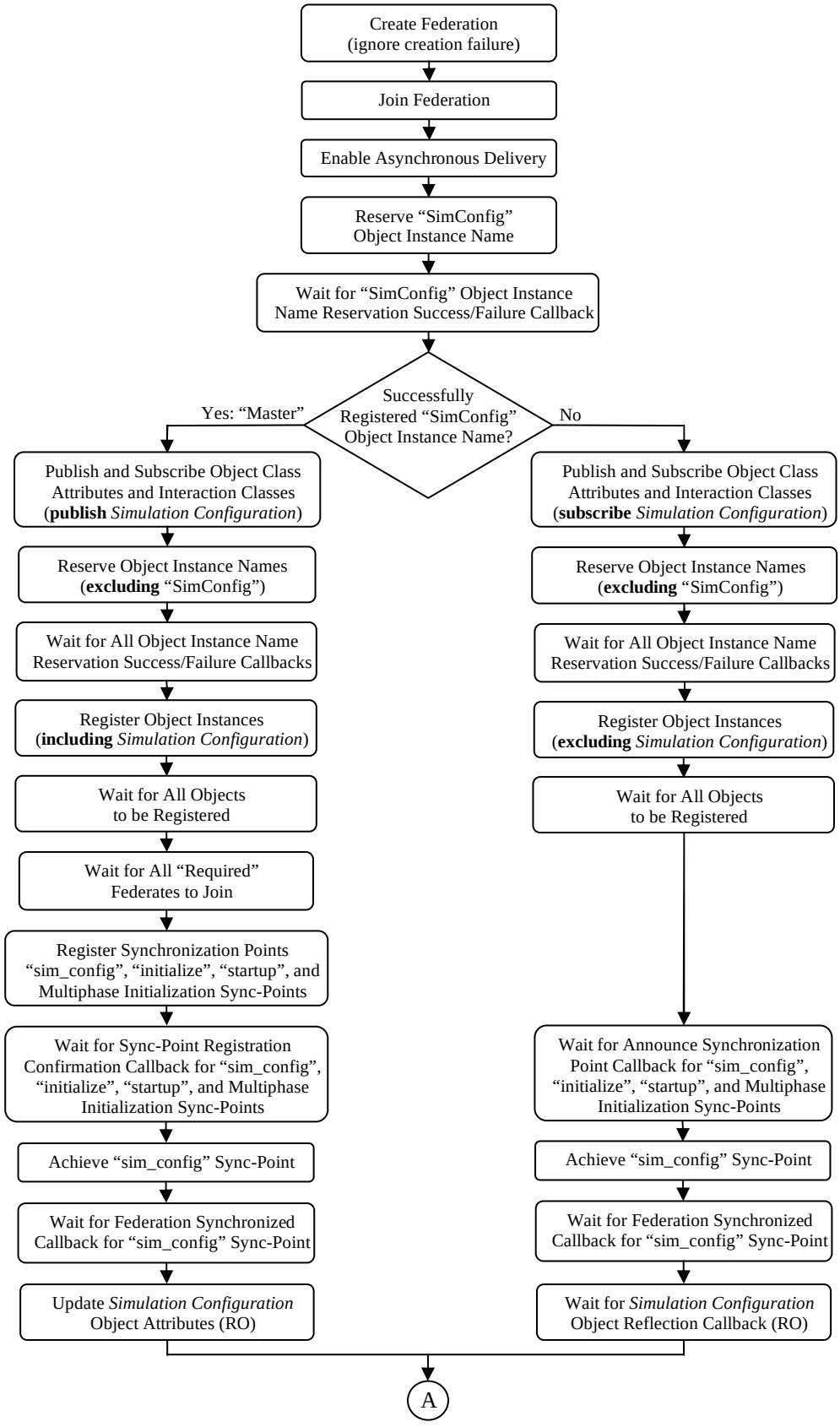
```
string name("");
int    size = attr_handle_value.size();
char * data = (char *) attr_handle_value.data();

for ( int i = 5; i < size; i += 2 ) {
   name.append( data+i, 1 );
}
```

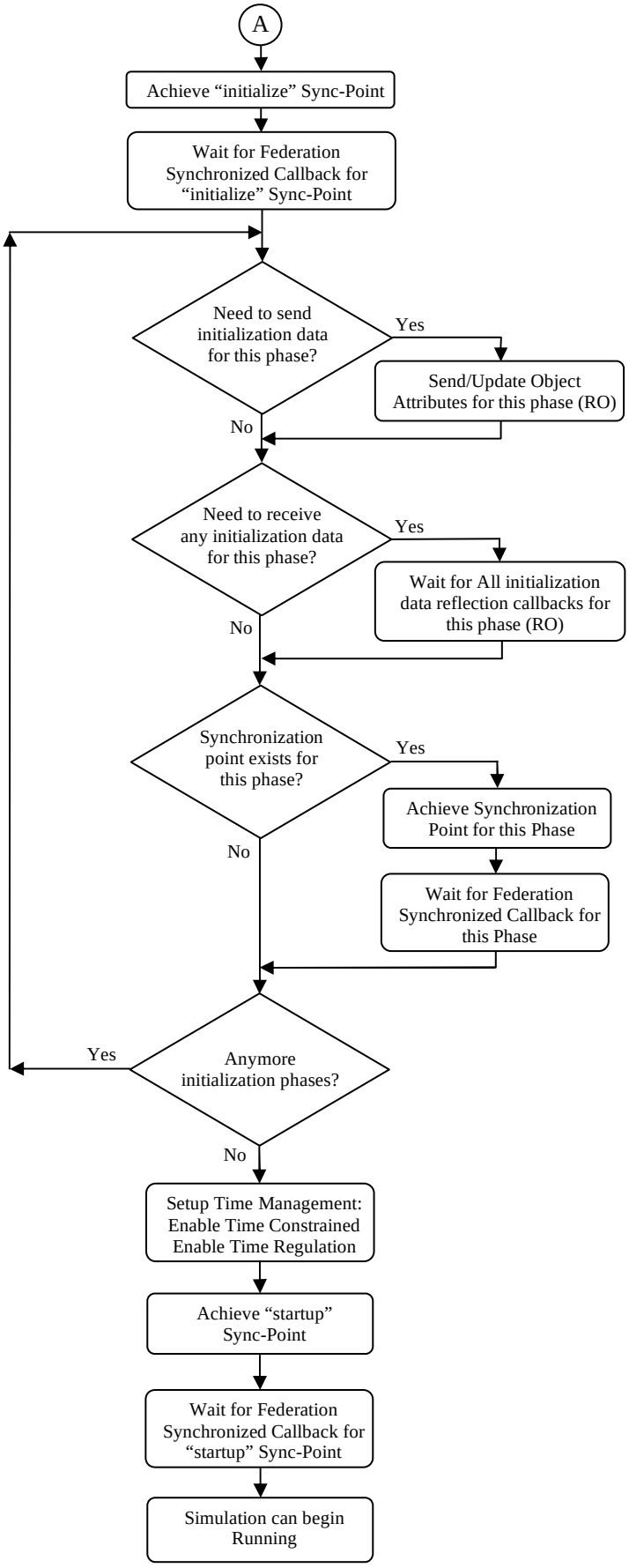
This assumes that the string stored in the `HLAunicodeString` only contains characters that are represented in the ASCII character set. If an expanded character set (e.g., Cyrillic or Kanji characters) is stored, this decoding will not work.

A.4 Time Objects

The HLA uses two abstract object classes, **LogicalTime**, and **LogicalTimeInterval**, to manage time. The Pitch RTI includes two subclasses that can be used for time encoded as doubles: **LogicalTimeDouble** and **LogicalTimeIntervalDouble**. The DSES project has created their own class versions, called **DoubleTime** and **DoubleInterval** that use the same encoding as the Pitch RTI objects, but add more programmer friendly access, such as overloaded comparison functions and access to the time as either a floating-point time in seconds or as an integer time in microseconds.

B Multiphase Initialization Process Flowchart




A
Achieve “initialize” Sync-Point
Wait for Federation Synchronized Callback for “initialize” Sync-Point
Need to send initialization data for this phase?
Yes
Send/Update Object Attributes for this phase (RO)
No
Need to receive any initialization data for this phase?
Yes
Wait for All initialization data reflection callbacks for this phase (RO)
No
Synchronization point exists for this phase?
Yes
Achieve Synchronization Point for this Phase
Wait for Federation Synchronized Callback for this Phase
No
Anymore initialization phases?
Yes
No
Setup Time Management: Enable Time Constrained Enable Time Regulation
Achieve “startup” Sync-Point
Wait for Federation Synchronized Callback for “startup” Sync-Point
Simulation can begin Running