View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

Integrated Mission Simulation (IMSim)

Multiphase Initialization Design with Late Joiners, Rejoiners and Federation Save & Restore

Daniel E. Dexter
NASA

Tony E. Varesic
L-3 Communications, Inc.

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

Date: May 7, 2015
Revision: 6

National Aeronautics and Space Administration

Acknowledgments

The Integrated Mission Simulation (IMSim) multiphase initialization design is the result of collaboration among the following individuals: Mike Blum, Dr. Edwin Z. Crues, Dan Dexter, Jim Gibson, David Hasan, Joe Hawkins, Robert Phillips, Mark Ricci, Danny Strauss, Paul Sugden, and Tony Varesic.

Table of Contents

1 Introduction 6
2 Requirements 6
3 Overview 7
4 Compatibility with a previous version 9
5 Federation Creation 9
5.1 Create the Federation 10
5.2 Join the Federation 10
5.3 Enable Asynchronous Delivery 10
6 Master Federate Simulation Configuration 10
6.1 Wait for All "Required" Federates to Join 11
6.2 Register "sim_config_v2" Synchronization Point with joined Federates only 15
6.3 Publish and Subscribe 15
6.4 Reserve Object Instance Names 16
6.5 Wait for All Object Instance Name Reservations 16
6.6 Register Object Instances 16
6.7 Wait for All Required Objects to be Registered 17
6.8 Register Synchronization Points for joined federates only 18
6.9 Wait for Synchronization Point Registration Confirmations 19
6.10 Achieve "sim_config_v2" Synchronization Point 19
6.11 Wait for "sim_config_v2" Federation Synchronization 19
6.12 Update *Simulation Configuration* Object Attributes 20
7 Non-Master Federate Simulation Configuration 20
7.1 Publish and Subscribe 20
7.2 Reserve Object Instance Names 21
7.3 Wait for All Object Instance Name Reservations 21
7.4 Register Object Instances 21
7.5 Wait for All Required Objects to be Registered 21
7.6 Wait for Announcement of Synchronization Points 21
7.7 Achieve "sim_config_v2" Synchronization Point 22
7.8 Wait for "sim_config_v2" Federation Synchronization 22
7.9 Wait for *Simulation Configuration* Object Reflections 22
8 Federate Initialization 23
8.1 Achieve "initialize_v2" Synchronization Point 23
8.2 Wait for "initialize_v2" Federation Synchronization 23
8.3 Send Initialization Data for the Current Phase 23
8.4 Receive Initialization Data for the Current Phase 24
8.5 Process Synchronization Point for the Current Phase 24
8.6 Determine if Additional Initialization Phases Exist 24
8.7 If "master" federate, register the "initialization_complete_v2" Synchronization Point 25
8.8 Request the number of and the names of all joined federates from the MOM 25
8.9 Setup Time Management 28
8.9.1 Time Frames 28
8.9.2 Real Time 28
8.9.3 Simulation Time 28

8.9.4 RTI Time 28
8.9.5 Update Time 29
8.9.6 Time Management 29
8.10 Achieve "startup_v2" Synchronization Point 29
8.11 Wait for "startup_v2" Federation Synchronization 29
8.12 Begin simulation execution 29
9 Late-Joiner Initialization 29
9.1 Subscribe to *Simulation Configuration* Object Class Attributes 30
9.2 Wait for *Simulation Configuration* Object to be Registered 30
9.3 Request *Simulation Configuration* Object update 30
9.4 Wait for *Simulation Configuration* Object Reflection Callback 31
9.5 Publish and Subscribe 31
9.6 Wait for Subscribed Object Class Discovery Callback 31
9.7 Was the instance attribute object discovered? 33
9.8 The instance attribute object was discovered (federate is rejoining the federation) 33
9.8.1 Re-acquire ownership of all published attributes 33
9.8.2 Wait until ownership transfer is complete 35
9.9 The instance attribute object was not discovered (federate is joining the federation for the first time) 37
9.9.1 Reserve Object Instance Names 37
9.9.2 Wait for All Object Instance Name Reservations 37
9.9.3 Register Object Instances 37
9.10 Wait for All Required Objects to be Registered 37
9.11 Request the number of and the names of all joined federates from the MOM 39
9.12 Is HLA Time Management Being Used? 39
9.13 Setup Time Management 39
9.14 Query the GALT 39
9.15 Time Advance Request to the GALT 39
9.16 Join the simulation execution, already in progress 39
10 Federate Execution 39
11 How to configure the Pitch RTI for federation save and restore 40
12 How to restore a saved federation (the first federate in the federation) 40
12.1 Read the '.running_feds' file and update list of "Required" federates 41
12.2 Wait for the required federates to join the federation 41
12.3 Look for any "unauthorized" federates 41
12.4 Load the checkpoint file 41
12.5 Request federation restore 41
12.6 Wait for restore request callback 41
12.7 Wait for the "restore begun" callback 42
12.8 Wait for the "ready to restore" callback 43
12.9 Inform RTI of the success/failure of the federate restore 43
12.10 Wait until the federation is restored 44
12.11 Request the federate handles from RTI 46
12.12 Register "startup_v2"sync point with all joined federates 48
12.13 Wait for the announcement of "startup_v2"sync point 48
12.14 Restart the federate (including setting up all RTI handles) 48
12.14.1 Restore sync points 48

12.14.2 Restore pending interactions 48
12.14.3 Restore objects and their attributes 48
12.14.4 Interactions and their parameters 50
12.14.5 Any pending ownership transfer commands 51
12.15 Achieve and wait for the "startup_v2" sync point 51
12.16 Resume execution of the federation 51
13 How to restore a saved federation (not the first federate in the federation) 51
13.1 Load the checkpoint file 52
13.2 Wait for the "restore begun" callback 52
13.3 Wait for the "ready to restore" callback 52
13.4 Inform RTI of the success/failure of the federate restore 52
13.5 Wait until the federation is restored 52
13.6 Wait for the announcement of "startup_v2"sync point 52
13.7 Restart the federate (including setting up all RTI handles) 52
13.8 Achieve and wait for the "startup_v2"sync point 52
13.9 Resume execution of the federation 52
14 Initiating a Federation Save 52
15 Saving the federation 55
15.1 Process / emulate the *Freeze Interaction* 55
15.2 Suspend / freeze execution after the completion of frame 55
15.3 Acknowledge the "FEDSAVE_v2" federation save sync point 56
15.4 Wait for federation to synchronize on the "FEDSAVE_v2" sync point 56
15.5 Request a federation save from the RTI 56
15.6 Wait until the RTI callback informs us to save our data 57
15.7 Turn off all data exchange and interaction processing 58
15.8 Save running federate information into an external file 58
15.9 Convert all interactions, ownership transfers and sync points into save-able format 59
15.10 Signal the RTI that this federate has begun saving 60
15.11 Write the checkpoint file 61
15.12 Signal the RTI that the federate save has completed saving 62
15.13 Wait for all federates to complete their save 63
15.14 Decode and report reason(s) for federation save failure 63
15.15 Restart data exchange & interaction processing 64
15.16 Resume execution using the "FEDRUN_v2" synchronization point 64
15.16.1 Registering the "FEDRUN_v2" synchronization point 64
15.16.2 Achieve the "FEDRUN_v2" synchronization point 64
15.16.3 Wait for the federation to be synchronized on "FEDRUN_v2" 65
16 How to resign from the federation with the intention of rejoining 65
17 References 66
A Subtle Points in Working with IEEE 1516 67
A.1 Calling the RTI and Threads 67
A.2 Wide Strings 67
A.3 HLAunicodeString 67
A.4 HLAhandle 68
A.5 Time Objects 69
B Multiphase Initialization Process Flowchart 70
C Federation Save Process Flowchart 74

Table of Figures

Figure 1 IMSim Multiphase Initialization High-Level Representation 8
Figure 2 IMSim Federation Save High-Level Representation........ 53
Figure 3 HLAunicodeString Format 68
Figure 4 "CEV" in the HLAunicodeString Format 68
Figure 5 HLAhandle Format........ 69
Figure 6 Example of the HLAbyte Format 69

1 Introduction

This document describes the design of the Integrated Mission Simulation (IMSim) federate multiphase initialization process. The main goal of multiphase initialization is to allow for data interdependencies during the federate initialization process.

IMSim 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. Please note the C++ code samples shown in this document are for the IEEE 1516-2000 standard.

Revision History:

Revised by:	Revision:	Date:	Description:
Tony E. Varesic	1	February 2010	Switched ‘retrieval of federate names from the MOM’ and ‘setup time management’ to avoid a race condition; updated federation save design to be based on the HLA logical time (not when the *Freeze Interaction* is received as was in the original design); code samples cleanup.
Tony E. Varesic	2	April 2010	Added the ability to rejoin a running federation into the federation design. Updated Section 9; added section on how to resign federate (Section 16); renumbered existing Section 16 to 17; updated Appendix B.
Daniel E. Dexter	3	December 2014	Updated flowchart in Appendix B to show the paths for nominal initialization, checkpoint restart, and late joiners and rejoining federates. Documented FEDRUN_v2 synchronization point.
Daniel E. Dexter	4	February 2015	Updated flowchart in Appendix B to show that late joiners or rejoining federates are not supported if HLA time management is not used. Documented the steps a late joining federate must take to catch-up to the logical time of the other running federates.
Daniel E. Dexter	5	May 2015	The data type for the “time” parameter of the *Freeze* Interaction should have been HLAinteger64BE. Updated flowchart in Appendix B to support late joining and rejoining federates that do not used HLA time management but use Common Timing Equipment (CTE).
Daniel E. Dexter	6	May 2015	Included the *Freeze* interaction FOM Module in the text.

2 Requirements

IMSim multiphase initialization has the following requirements:

1. IMSim 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. The condition(s) for a federation save, if applicable, shall be predetermined and the triggering of this functionality be programmed into at least one federate ahead of time.

3 Overview

The IMSim 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, and whether the federation is executing when the federate joined the federation. Figure 1 below shows the relationship of these stages, which is a high-level representation of the IMSim multiphase initialization process flowchart shown in Appendix B.

The multiphase initialization scheme also allows the restart of a federation from a checkpoint set. The federation restore would occur instead of the federation initialization stage in the master / non-master simulation configuration; the restoring of a federation during a federation execution has yet to be implemented.

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

1. Federation Creation
2. Simulation Configuration
   a. Master Federate Specific Configuration
      If the federation restore command was given upon startup
      i. Restore “Master” Federate Specific Configuration
      Otherwise,
      ii. Federate Initialization
   b. Non-Master Federate Specific Configuration
      If the federation restore command was given upon startup
      i. Restore “Non-Master” Federate Specific Configuration
      Otherwise,
      ii. Federate Initialization
   c. Late-Joiner Federate Specific Configuration
3. Federate Execution

The IMSim multiphase initialization process is modular allowing functionality to be added as needed. At a minimum the nominal initialization paths shown in green in the IMSim multiphase initialization flowchart in Appendix B needs to be implemented. Support for checkpoint restart can be added by implementing the orange checkpoint paths shown in the flowchart in Appendix B. Support for late joining and rejoining federates can be added by implementing the blue path shown in the flowchart in Appendix B.

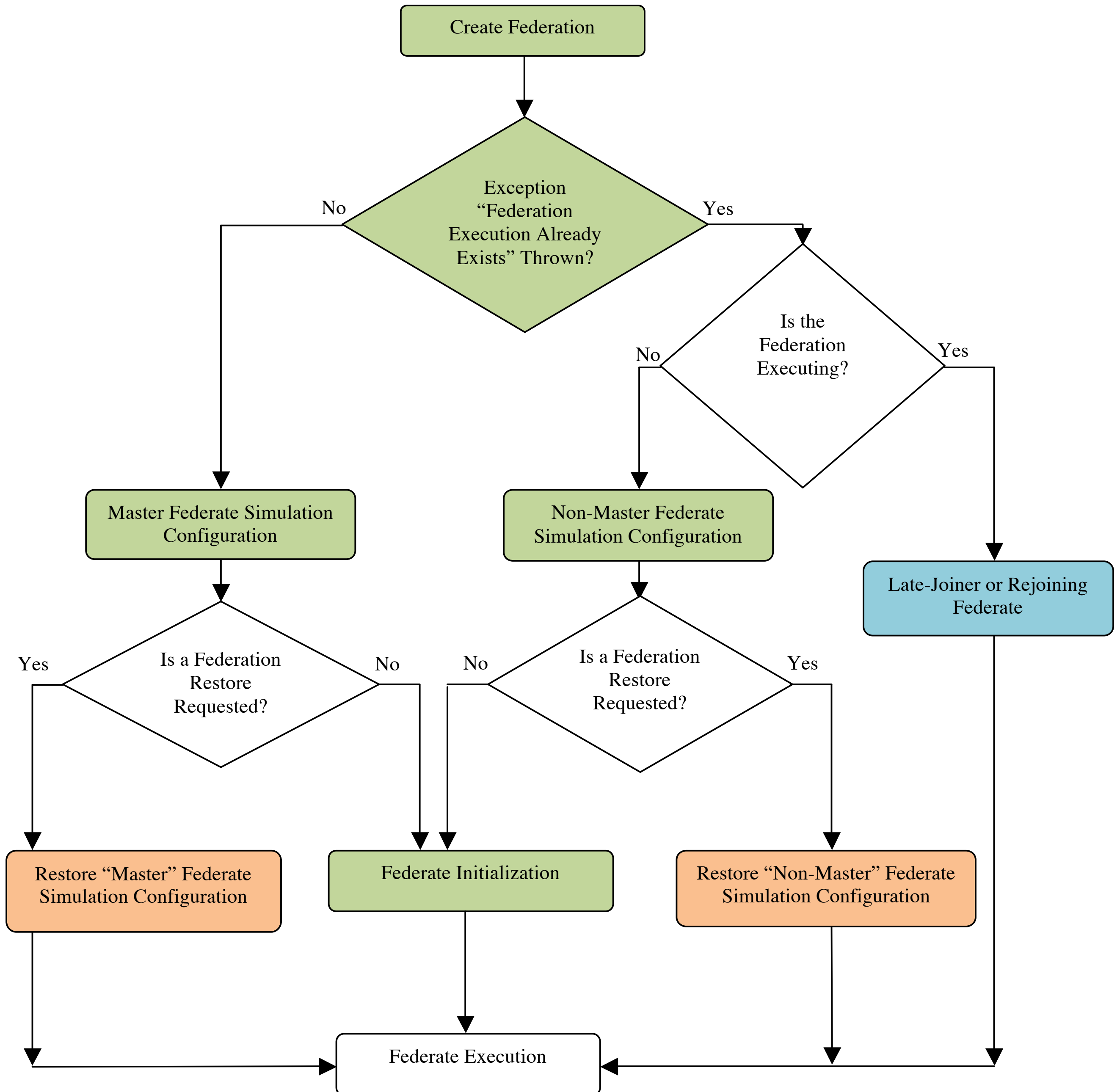


Figure 1 IMSim Multiphase Initialization High-Level Representation

This document will refer to a federate by a few names, as you may have already seen. The following is a breakdown of what they are and what they mean:

- Master – the federate which created the federation; it will drive the multiphase initialization process in one-way or another.
- Non-Master – the federate that did not succeed in creating the federation, which means they are not the first federate.
- Late-Joiner – the federate that joined the federation after the required federates have completed initialization and are running.
- Rejoiner – the federate that has resigned from the federation and is now trying to rejoin.

4 Compatibility with a previous version

Since the ability for federates to join an already-running federation and the ability to save / restore the federation were added into IMSim multiphase initialization process, henceforth referred to as "multiphase initialization version 2", backwards compatibility with the previous implementation of the multiphase initialization (DSES Multiphase Initialization Design [3], henceforth referred to as "multiphase initialization version 1") is not possible. All federates must use the same version of the multiphase initialization. Otherwise, the federation will fail to start properly.

If the user attempts to start up the federation with differing versions of the multiphase initialization, the federation startup will always end up in deadlock, i.e. failing to startup properly, exhibiting one of these failure scenarios:

1) If multiphase initialization version 1 starts the federation and multiphase initialization version 2 federate joins the federation, the multiphase initialization version 1 federate will be stuck in the spin-lock loop waiting for all objects to join the federation while the multiphase initialization version 2 federate will go into a spin-lock loop immediately after joining the federation when determining if it is a restore or late-joining federate. Since version 1 of the multiphase initialization does not support late-joiner federates, it will not send the late-joiner determining sync point to the federation so the multiphase initialization version 2 federate will be stuck in a spin-lock loop looking for this late-joiner determining sync point indefinitely.
2) If multiphase initialization version 2 starts the federation and a multiphase initialization version 1 federate joins the federation, both federates will go into a spin-lock loop waiting for 'all initialization sync point registrations'. Since the initialization sync point names differ between the versions of the multiphase initialization, both federates will be stuck in this spin-lock loop waiting for their expected sync point names indefinitely.

If the user attempts to restore a federation with differing versions of the multiphase initialization, the federation restore will end up in deadlock exhibiting one of the following scenarios:

1) If multiphase initialization version 1 starts the federation and a multiphase initialization version 2 federate joins the federation in hopes to continue a federation restore, the multiphase initialization version 1 federate will be stuck in the spin-lock loop waiting for all objects to join the federation while the multiphase initialization version 2 federate will go into a spin-lock loop immediately after joining the federation when trying to determine if this is a restore or late-joining federate. Since version 1 of the multiphase initialization does not support federation restores, it will be stuck in this spin-lock loop indefinitely, waiting for all object to register.
2) If multiphase initialization version 2 starts the federation in 'federation restore' mode and multiphase initialization version 1 federate joins the federation, the multiphase initialization version 1 federate will be stuck in the spin-lock loop waiting for all objects to get registered with the federation while the multiphase initialization version 2 federate will be stuck in the spin-lock loop waiting for the 'federationRestoreComplete' callback from the RTI.

5 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.

All federates execute this section before branching off into their respective paths through the initialization process.

5.1 Create the Federation
5.2 Join the Federation
5.3 Enable Asynchronous Delivery

5.1 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.

5.2 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 );
```

5.3 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();
```

6 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 second branch from the left of the multiphase initialization flowchart shown in Appendix B.

6.1 Wait for All “Required” Federates to Join
6.2 Register “sim_config_v2” Synchronization Point for joined Federates only
6.3 Publish and Subscribe
6.4 Reserve Object Instance Names
6.5 Wait for All Object Instance Name Reservations
6.6 Register Object Instances
6.7 Wait for All Required Objects to be Registered
6.8 Register Synchronization Points for joined Federates only
6.9 Wait for Synchronization Point Registration Confirmations
6.10 Achieve “sim_config_v2” Synchronization Point
6.11 Wait for “sim_config_v2” Federation Synchronization

6.12 Update Simulation Configuration Object Attributes

6.1 Wait for All "Required" Federates to Join

The initialization scheme utilizes the RTI to push synchronization points to all federates who join at federation startup, thus, we need to ensure that all required federates have joined before proceeding. This way, a late joiner federate will not receive any synchronization points, which are used in the initialization stage of the federation creation.

Since the master federate has successfully created the federation in section 5.1, the next step is to ensure that all other required federates have 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" );

// Get the federate ID for the federate object
federate_id_handle = rtiAmbassador->getAttributeHandle(
                        federate_type_id, L"HLAfederateHandle" );


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

// 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 PitchRTI
// to force an immediate data update.
try {
   attributes.insert( federate_name_type_id );
   attributes.insert( federate_id_handle );
   rtiAmbassador->requestAttributeValueUpdate( federate_type_id,
                                               attributes,
                                               UserSuppliedTag( 0, 0 ) );
} catch( RTI::exception & e ) {
   wcerr << L"Update request failed!" << e.what() << 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 id and 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 );
        myfederate->add_discovered_object_federate_instance_id(
                     theObject, theObjectInstanceName );
    } 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 `add_discovered_object_federate_instance_id()` function is used to maintain the list of MOM federate names.

```
void MyFederate::add_discovered_object_federate_instance_id (
   ObjectInstanceHandle instance_id,
   wstring const & theObjectInstanceName )
{
   discovered_object_federate_name_map[ instance_id ] =
                                       theObjectInstanceName;
}
```

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;
   }
}
```

Additionally, a collection of all joined federate IDs, called `joined_federate_set`, will be populated with each federate's ID as it is discovered. This collection, which is of RTI_FederateHandleSet type, will be used to supply the RTI the list of federate IDs, which are to receive synchronization points used in the multiphase initialization process.

A federate ID is found in the map of attribute handle values by using the attribute handle `federate_id_handle`, which corresponds to the `HLAfederateHandle` attribute that was subscribed to. The federate ID, in the `HLAhandle` format, is decoded using the following code which uses the technique described in Appendix A, section A.4 below.

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

   // Find the FederateHandle attribute for the given MOM federate handle.
   attr_iter = values->find( federate_id_handle );

   // Determine if we have a federate handle attribute.
   if ( attr_iter != values->end() ) {

      int num_bytes = attr_iter->second.size();
      int * data = (int *)attr_iter->second.data();

      // Do a sanity check on the size.
      if ( num_bytes != 8 ) {
         fatalError( "extract_federate_id():%d Unexpected number of bytes in "
                     " the FederateHandle because the byte count is %d"
                     " but we expected 8!", __LINE__, num_bytes );
      }

      // The HLAfederateHandle has the HLAhandle datatype which is has the
      // HLAvariableArray encoding with an HLAbyte element type.
      //  0 0 0 4 0 0 0 2
      //  ---+--- | | | |
      //     |    ---+---
      // #elem=4  fedID = 2
      //
      // Determine if we need to byteswap or not since the FederateHandle
      // is in Big Endian. First 4 bytes (first 32-bit integer) is the number
      // of elements.
      int fed_id = is_transmission_byteswap( THLA_BIG_ENDIAN ) ?
                      byteswap_int( data[1] ) : data[1];

      // Add this FederateHandle to the set of joined federates.
      joined_federate_set.insert( FederateHandle( fed_id ) );
}
```

6.2 Register "sim_config_v2" Synchronization Point with joined Federates only

The master federate is responsible for registering the "sim_config_v2" synchronization point with the joined federates only, contained in `joined_federate_set`. The following call creates the synchronization points needed for the federate multiphase initialization process.

```
rtiAmbassador->registerFederationSynchronizationPoint( L"sim_config_v2",
       UserSuppliedTag( 0, 0 ), joined_federate_set );
```

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_v2" ) == 0 ) {
      myfederate->set_sim_config_sp_exists( true );
   }
   else if ( sp_label.compare( L"initialize_v2" ) == 0 ) {
      myfederate->set_initialize_sp_exists( true );
   }
   else if ( sp_label.compare( L"startup_v2" ) == 0 ) {
      myfederate->set_startup_sp_exists( true );
   }
   else if ( myfederate->is_multiphase_sp( sp_label ) ) {
      myfederate->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 );
}
```

6.3 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.

6.4 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 6.2 above. The names to be used in a federation should be agreed upon in advance.

6.5 Wait for All Object Instance Name Reservations

The federate must wait for the name reservation success or failure to be acknowledged with an asynchronous callback to the federate ambassador. This includes the "SimConfig" object published in section 6.3. 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 ) {
       myfederate->set_master( true );
   }
   myfederate->set_name_reserved( instance_name );
}
void MyFedAmbassador::objectInstanceNameReservationFailed( instance_name ) {
    if ( instance_name.compare( L"SimConfig" ) == 0 ) {
       myfederate->set_master( false );
       myfederate->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 );
}
```

6.6 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.

```
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
```

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)
{
   saveInstance( object_id, instance_id, instance_name );
}
```

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”.

6.7 Wait for All Required Objects to be Registered

Next the master federate waits for all the required object instances it will be using during the course of the simulation to be registered with the RTI. What makes an object instance ‘required’ is that it is locally owned by a federate required for federation startup.

A required federate can subscribe to a late-joiner federate’s object instance as long as the required federate identifies the subscribed-to object instance as ‘not required’. The data published by the late-joiner may not start streaming into the required federate until the next frame or a few frames after the late-joiner joins the federation and is discovered by a federate.

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. All discovered objects also should be marked as required so that in the event of a federation save, its correct state would be saved thus indicating that this object instance would, along with the joined federate, be required when the federation is restored.

```
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_required() ) && ( ! object[n].is_registered() ) ) {
```

```
         any_unregistered_obj = true;

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

   usleep( 100 );
} while ( any_unregistered_obj );
```

6.8 Register Synchronization Points for joined federates only

The master federate is responsible for registering the “initialize_v2”, and “startup_v2” synchronization points as well as all the multiphase initialization synchronization points, if any exist, with all joined federates contained in `joined_federate_set`. The following calls create the synchronization points needed for the federate multiphase initialization process.

```
rtiAmbassador->registerFederationSynchronizationPoint( L"initialize_v2",
       UserSuppliedTag( 0, 0 ), joined_federate_set );

rtiAmbassador->registerFederationSynchronizationPoint( L"startup_v2",
       UserSuppliedTag( 0, 0 ), joined_federate_set );

// 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 ), joined_federate_set );
}
```

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.

```
void MyFedAmbassador::synchronizationPointRegistrationSucceeded(
       wstring sp_label )
{
   if ( sp_label.compare( L"sim_config_v2" ) == 0 ) {
      myfederate->set_sim_config_sp_exists( true );
   }
   else if ( sp_label.compare( L"initialize_v2" ) == 0 ) {
      myfederate->set_initialize_sp_exists( true );
   }
   else if ( sp_label.compare( L"startup_v2" ) == 0 ) {
      myfederate->set_startup_sp_exists( true );
   }
   else if ( myfederate->is_multiphase_sp( sp_label ) ) {
      myfederate->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.

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

6.9 Wait for Synchronization Point Registration Confirmations

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

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

The `all_sp_exist()` function returns true once the "sim_config_v2", "initialize_v2", "startup_v2", 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;
}
```

6.10 Achieve "sim_config_v2" 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_v2" synchronization point only after all required federates are joined, and all object instances required for initialization have been created.

```
rtiAmbassador->synchronizationPointAchieved( L"sim_config_v2" );
```

6.11 Wait for "sim_config_v2" 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_v2" ) == 0 ) {
      myfederate->set_sim_config_sp_synchronized( true );
   }
   else if ( sp_label.compare( L"initialize_v2" ) == 0 ) {
      myfederate->set_initialize_sp_synchronized( true );
   }
   else if ( sp_label.compare( L"startup_v2" ) == 0 ) {
      myfederate->set_startup_sp_synchronized( true );
   }
   else if (myfederate->is_multiphase_sp( sp_label ) ) {
      myfederate->set_multiphase_sp_synchronized( sp_label, true );
   }
}
```

The federate will wait in a loop until the "sim_config_v2" synchronization point has been synchronized for the federation.

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

6.12 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, UserSuppliedTag( 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() );
```

7 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 middle branch of the multiphase initialization flowchart shown in Appendix B.

7.1 Publish and Subscribe
7.2 Reserve Object Instance Names
7.3 Wait for All Object Instance Name Reservations
7.4 Register Object Instances
7.5 Wait for All Required Objects to be Registered
7.6 Wait for Announcement of Synchronization Points
7.7 Achieve "sim_config_v2" Synchronization Point
7.8 Wait for "sim_config_v2" Federation Synchronization
7.9 Wait for *Simulation Configuration* Object Reflections

7.1 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 6.3 are followed with the exception that the *Simulation Configuration* object is only subscribed to and not published.

7.2 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 6.4 above. The names to be used in a federation should be agreed upon in advance.

7.3 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 6.5 above, but with a reserved status flag specific to each object instance name.

7.4 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 6.6 above are followed with the exception that an instance for the *Simulation Configuration* object is not created.

7.5 Wait for All Required 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 required object instances it will be using to be registered by following procedures in section 6.7 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. Unless a federate is required for simulation startup, we must not wait for its corresponding data object instance to show up because it will not start being reflected until shortly after the non-required federate joins the federation.

7.6 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_v2" ) == 0 ) {
      myfederate->set_sim_config_sp_exists( true );
   }
   else if ( sp_label.compare( L"initialize_v2" ) == 0 ) {
      myfederate->set_initialize_sp_exists( true );
   }
   else if ( sp_label.compare( L"startup_v2" ) == 0 ) {
      myfederate->set_startup_sp_exists( true );
   }
   else if ( myfederate->is_multiphase_sp( sp_label ) ) {
      myfederate->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 );
```

```
}
```

7.7 Achieve “sim_config_v2” Synchronization Point

Next the non-master federate achieves the “sim_config_v2” synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L"sim_config_v2" );
```

7.8 Wait for “sim_config_v2” Federation Synchronization

The non-master federate follows the same procedures outline in section 6.11 above to wait for the “sim_config_v2” federation synchronization.

7.9 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 );
   }
```

```
}
```

8 Federate Initialization

The following steps comprise the “Federate Initialization” stage of the multiphase initialization process shown in Figure 1, which corresponds to “leg H” located on the second page of the multiphase initialization flowchart shown in Appendix B.

8.1 Achieve “initialize_v2” Synchronization Point
8.2 Wait for “initialize_v2” Federation Synchronization
8.3 Send Initialization Data for the Current Phase
8.4 Receive Initialization Data for the Current Phase
8.5 Process Synchronization Point for the Current Phase
8.6 Determine if Additional Initialization Phases Exist
8.7 If “master” federate, register the“initialization_complete_v2” Synchronization Point
8.8 Request the number of and the names of all joined federates from the MOM
8.9 Setup Time Management
8.10 Achieve “startup_v2” Synchronization Point
8.11 Wait for “startup_v2” Federation Synchronization
8.12 Begin simulation execution

8.1 Achieve “initialize_v2” Synchronization Point

The federate achieves the “initialize_v2” synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L"initialize_v2" );
```

8.2 Wait for “initialize_v2” Federation Synchronization

The federate should wait in a blocking loop until the “initialize_v2” synchronization point has been synchronized for the federation. See section 6.11 for the Federate Ambassador federation synchronized callback.

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

8.3 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 ) );
   }
}
```

8.4 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 );
      }
   }
}
```

8.5 Process Synchronization Point for the Current Phase

Next the federate achieves the user-defined, multiphase, synchronization points and waits for federation synchronization if needed 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() ) {

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

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

8.6 Determine if Additional Initialization Phases Exist

The federate shall keep looping through the steps of sending (section 8.3), receiving (section 8.4), and processing synchronization points (section 8.5) 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 8.7) step.

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

   receive_init_data_for_phase();

   process_sync_pt_for_phase();
```

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

8.7 If "master" federate, register the "initialization_complete_v2" Synchronization Point

If we are the master federate, we need to register an "initialization_complete_v2" synchronization point with the RTI. This sync point indicates that the initialization stage of the federation has been completed and, thus, it will be used at the determining factor when a federate joins the federation whether it is a late-joiner. If a federate joins the federation after this synchronization point has been registered, they automatically follow the late-joiner path of the multiphase initialization process (far right leg of the Multiphase Initialization Process Flowchart in Appendix B).

This is a marker synchronization point, thus, it shall not be used as a federation pause point.

8.8 Request the number of and the names of all joined federates from the MOM

Before the federate starts executing, it should request the names of all federates currently joined in the federation from the MOM so that when a federation save is requested, we have a current list of joined federates. This list will be used as the "required federates" list when the federation is restored.

Each federate shall request from the MOM the number of federates currently joined in the federation and request from the MOM the names of all joined federates. Section 15.8 will save this information into a file for use in a subsequent restore of the federation.

All federates should make this request only once. In order to keep the federate names current, we shall not unsubscribe from this 'requestAttributeValueUpdate', so that when a federate joins (indicated by the `MyFedAmbassador::discoverObjectInstance()` RTI callback routine) or resigns (indicated by the `MyFedAmbassador::deleteObjectInstance()` RTI callback routine) from the federation, these callbacks would signal us to add or remove, respectively, a federate name from the list of known federates. However, we should unsubscribe to the number federates since we can keep that count current ourselves.

Trigger this code in each federate before it turns on time management:

```
void MyFederate::load_running_federates()
{
   MOM_federation_class_handle = RTIAmbassador->getObjectClassHandle(
                                              L"HLAmanager.HLAfederation" );
   MOM_federate_count_handle = RTIAmbassador->getAttributeHandle(
                                            MOM_federation_class_handle,
                                            L"HLAfederatesInFederation" );

   AttributeHandleSet fedMomAttributes;
   fedMomAttributes.insert( MOM_federate_count_handle );

   RTIAmbassador->subscribeObjectClassAttributes( MOM_federation_class_handle,
                                                  fedMomAttributes,
                                                  true );

   AttributeHandleSet requestedAttributes;
   requestedAttributes.insert( MOM_federate_count_handle );

   RTIAmbassador->requestAttributeValueUpdate( MOM_federation_class_handle,
                                               requestedAttributes,
                                               UserSuppliedTag( 0, 0 ) );

   // wait until the RTI callback provides us with a count
   while ( running_feds_count <= 0) {
      // Sleep a little while to wait for the information to update.
```

```
      usleep( 1000 );
   }

   // We are done with the MOM interface to unsubscribe from the classes we used.
   RTIAmbassador->unsubscribeObjectClass( MOM_federation_class_handle );

   MOM_federate_class_handle = RTIAmbassador->getObjectClassHandle(
       L"HLAobjectRoot.HLAmanager.HLAfederate" );
   MOM_federate_name_handle = RTIAmbassador->getAttributeHandle(
       MOM_federate_class_handle, L"HLAfederateType" );
   MOM_federate_handle = RTIAmbassador->getAttributeHandle(
       MOM_federate_class_handle, L"HLAfederateHandle" );

   AttributeHandleSet fedMomAttributes;
   fedMomAttributes.insert( MOM_federate_name_handle );
   fedMomAttributes.insert( MOM_federate_handle );

   // Subscribe the MOM object attributes.
   RTIAmbassador->subscribeObjectClassAttributes( MOM_federate_class_handle,
                                                  fedMomAttributes,
                                                  true );

   AttributeHandleSet requestedAttributes;
   requestedAttributes.insert( MOM_federate_name_handle );
   requestedAttributes.insert( MOM_federate_handle );

   // Request federate names from MOM
   RTIAmbassador->requestAttributeValueUpdate( MOM_federate_class_handle,
                                               requestedAttributes,
                                               UserSuppliedTag( 0, 0 ) );

   unsigned int joinedFedCount = 0;

   // Wait for all the required federates to join.
   all_federates_joined = false;
   while ( !all_federates_joined ) {
      // Sleep a little while to wait for more federates to join.
      usleep( 1000 );

      // Determine what federates have joined only if the joined federate
      // count has changed.
      if ( joinedFedCount != joined_federate_names.size() ) {
         joinedFedCount = joined_federate_names.size();

         if ( joinedFedCount >= (unsigned int) running_feds_count ) {
            all_federates_joined = true;
         }
      }
   }

   // execute a wait loop until the RTI callback provides us the information for
   // all running federates
   while ( federate_instances.size() < running_feds_count ) {
      usleep( 100 );
   }

   // Now, copy the new federate information into my data stores.

   // clear the running_feds data structure

   // update running_feds to contain name from federate_instances and
   // the MOM federate name from discovered_object_federate_name_map.
   map< ObjectInstanceHandle, wstring >::iterator map_iter;
   int loop = 0;
   for ( map_iter = federate_instances.begin();
         map_iter != federate_instances.end(); map_iter++ ) {
```

```
      running_feds[loop].name = map_iter->second.c_str();

      running_feds[loop].MOM_instance_name =
               discovered_object_federate_name_map[map_iter->first].c_str();

      // if the federate was running at the time of the checkpoint, it is now a
      // 'required' federate in the restore, regardless if it is was required
      // when the federation originally started up.
      running_feds[loop].required = 1;

      loop++;
   }
   // Do not un-subscribe to this MOM data; we DO want updates as federates
   // join / resign from the federation!
}
```

The number of federates would be sent to us via the `MyFedAmbassador::reflectAttributeValues()` callback. In this routine, it is necessary to add checks if the objectClass matches the `MOM_federation_class` we received from the RTI above and extract the number of federates by calling the following routine:

```
void MyFederate::get_number_of_federates(
   ObjectInstanceHandle                id,
   auto_ptr< AttributeHandleValueMap > values )
{
   AttributeHandleValueMap::iterator attr_iter;
   // load the first value from 'values'.
   attr_iter = values->begin();

   // Determine if were successful.
   if ( attr_iter != values->end() ) {

      string fed_id = id.getImplementation().c_str();
      // if this is a federate #, not an object #, process it...
      if ( fed_id == "1") {
         // Extract the size of the data and the data bytes.
         int * data = (int *)attr_iter->second.data();

         // The HLAfederatesInFederation has the HLAhandle datatype which is has
         // the HLAvariableArray encoding with a HLAbyte element type. The
         // entry is the number of elements, followed by that number of
         // HLAvariableArrays, each consisting of the # of bytes and an id.
         //  0 0 0 2 0 0 0 4 0 0 0 3 0 0 0 4 0 0 0 2
         //  ---+--- | | | | ---+--- | | | | ---+---
         //     |    ---+---    |    ---+---    |
         //   count   size   id #1    size   id #2
         //
         // The first 4 bytes (first 32-bit integer) is the number
         // of elements. WE ARE INTERESTED ONLY IN THIS VALUE!

         // Determine if we need to byteswap or not since the FederateHandle
         // is in Big Endian. First 4 bytes (first 32-bit integer) is the number
         // of elements.

         // save the count into running_feds_count
         running_feds_count = is_transmission_byteswap( THLA_BIG_ENDIAN ) ?
                              byteswap_int( data[0] ) : data[0];

      } else {
         cout << "ERROR: Federate::get_number_of_federates():"
              << "Received an id which is not a federate number: "
              << fed_id << endl;
      }
```

```
   } else {
        cout << "ERROR: Federate::set get_number_of_federates():"
             << "FederationHandle possibly is empty; size="
             << values->size() << endl;
   }
}
```

8.9 Setup Time Management

8.9.1 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.

8.9.2 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.

8.9.3 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.

8.9.4 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.

8.9.5 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.

8.9.6 Time Management

The following calls initialize the time management for a simulation that follows the IMSIM 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.

8.10 Achieve "startup_v2" Synchronization Point

Next the federate achieves the "startup_v2" synchronization point.

```
rtiAmbassador->synchronizationPointAchieved( L"startup_v2" );
```

8.11 Wait for "startup_v2" Federation Synchronization

The federate will wait in a loop until the "startup_v2" synchronization point has been synchronized for the federation. See section 6.11 for the Federate Ambassador federation synchronized callback.

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

8.12 Begin simulation execution

Once the federation has been synchronized on the "startup_v2" synchronization point the multiphase initialization is complete and the simulation can begin executing (see section 10).

9 Late-Joiner Initialization

The following steps comprise the "Late-Joiner" part of the multiphase initialization process show in Figure 1, which corresponds to the "Sync-Point 'initialization_complete_v2' Announced?", the far right branch of the multiphase initialization flowchart shown in Appendix B.

9.1 Subscribe to *Simulation Configuration* Object Class Attributes
9.2 Wait for *Simulation Configuration* Object to be Registered
9.3 Request *Simulation Configuration* Object update

9.4 Wait for *Simulation Configuration* Object Reflection Callback
9.5 Publish and Subscribe
9.6 Wait for Subscribed Object Class Discovery Callback
9.7 Was the instance attribute object discovered?
9.8 The instance attribute object was discovered (federate is rejoining the federation)
9.8.1 Re-acquire ownership of all published attributes
9.8.2 Wait until ownership transfer is complete
9.9 The instance attribute object was not discovered (federate is joining the federation for the first time)
9.9.1 Reserve Object Instance Names
9.9.2 Wait for All Object Instance Name Reservations
9.9.3 Register Object Instances
9.10 Wait for All Required Objects to be Registered
9.11 Request the number of and the names of all joined federates from the MOM
9.12 Is HLA Time Management Being Used?
9.13 Setup Time Management
9.14 Query the GALT
9.15 Time Advance Request to the GALT
9.16 Join the simulation execution, already in progress

A late-joiner federate is one which joins the federation after the federation has completed its initialization stage and is currently running. As previously mentioned in section 8.7, the “master” federate registers the “`initialization_complete_v2`” synchronization point with the RTI and it is this synchronization point that is sent to all late-joining federates when they join the federation. The presence of this synchronization point is to inform them that the federation has completed the initialization process and is currently executing so they must follow the “late-joiner” path of flowchart in Appendix B.

The federate also has the ability to rejoin a running federation if it resigns from the federation in the manner discussed in Section 16. This section will automatically detect if this federate has previously run in the federation and performs special steps in order to rejoin the federation.

9.1 Subscribe to *Simulation Configuration* Object Class Attributes

In order to get data from the *Simulation Configuration* object, we must first subscribe to it. The procedures outlined in section 6.3 are followed only for subscribing to the *Simulation Configuration* object; the publishing and subscribing of other objects will happen in section 9.5.

9.2 Wait for *Simulation Configuration* Object to be Registered

This step waits until the *Simulation Configuration* object is registered with the federation. Until it is registered, the federate executes a spin-lock loop.

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

9.3 Request *Simulation Configuration* Object update

Next, we request an update from the Simulation Configuration object. To accomplish this, we execute code that looks like the following:

```
// There needs to be at one remotely owned attribute that is subscribed to
// before we can request an update.
if ( ! any_remotely_owned_subscribed_attribute() ) {
   return;    }
```

```
// Create the set of Attribute handles we need to request an update for.
AttributeHandleSet attr_handle_set = AttributeHandleSet();

for ( int i = 0; i < attr_count; i++ ) {
   // Only include attributes that are remotely owned that are subscribed to.
   if ( attributes[i].is_remotely_owned() && attributes[i].is_subscribe() ) {
      attr_handle_set.insert( attributes[i].get_attribute_handle() );
   }
}

try {
   // supplying a UserDefinedTag(0,0) means it is a receive order (RO) request
   (void)RTIAmbassador->requestAttributeValueUpdate( instance_handle,
                                                     attr_handle_set,
                                                     UserSuppliedTag( 0, 0 ) );
   // Must free the memory
   attr_handle_set.clear();

} catch ( AttributeNotDefined &e ) {
   fatalError("request_attribute_value_update() attribute not defined for '%s'",
              name.c_str() );
} catch ( RestoreInProgress &e ) {
   fatalError("request_attribute_value_update() restore in progress for '%s'",
              name.c_str() );
} catch ( RTI1516_EXCEPTION &e ) {
   fatalError("request_attribute_value_update(): Exception: '%ls'",
              e.what() );
}
```

9.4 Wait for *Simulation Configuration* Object Reflection Callback

After we make a request for an update to the Simulation Configuration object, we wait for its callback by following the procedures outlined in section 7.9.

9.5 Publish and Subscribe

The late-joiner federate must publish and subscribe to all relevant HLA objects and interactions. The procedures outlined in section 6.3 are followed with the exception that the subscribing to the *Simulation Configuration* object is skipped since that was taken care of in section 9.1.

9.6 Wait for Subscribed Object Class Discovery Callback

Since the subscribe step above will cause the discovery of all instance attribute objects in the federation [based on RTI-assigned class type], this also means that the resigned federate's instance attribute object will be discovered if it exists. To avoid a potential race condition, we must pause the initialization process until this discovery has taken place. This is accomplished by add a blocking loop to make sure that the all of the objects are discovered before continuing the initialization process.

If the object discoveries have taken place but the instance attribute object for the resigned federate was not discovered, this means that the federate did not resign properly by either deleting itself when it resigned or the attribute ownership divestiture. Subsequent acquisition by other federate(s) failed for all of the attribute(s) – maybe because no other federate was configured to publish at least one attribute – which caused the instance attribute object to became a federation orphan. Emit an error message explaining what happened and terminate the federate.

This step is accomplished by adding and calling code similar to the following:

```
// do we have Simulation object(s) to interrogate?
if ( obj_count > 0 ) {
```

```
   // see if any object discoveries have occurred...
   int required_count = 0;
   int discovery_count = 0;
   bool create_HLA_instance_object_found = false;
   for ( int n = 0; n < obj_count; n++ ) {
      if ( objects[n].is_required() ) {
         required_count++;
      }
      if ( objects[n].is_instance_handle_valid() ) {
         discovery_count++;
         if ( objects[n].is_create_HLA_instance() ) {
            create_HLA_instance_object_found = true;
         }
      }
   }

   // if all of the required objects were discovered, exit immediately.
   if ( discovery_count == required_count ) {
      return;
   }

   // figure out how many objects have been discovered so far...
   //    if "still missing some objects other than the one for the rejoining
   federate,
   //        or missing some other object(s) but found the rejoining federate "
   if ( ( !create_HLA_instance_object_found &&
          ( discovery_count < ( required_count - 1 ) ) ) ||
        ( create_HLA_instance_object_found &&
          ( discovery_count < required_count ) ) ) {

      // block until some / all arrive.
      do {

         // Sleep for a little while to allow the RTI to trigger the object
         // discovery callbacks.
         usleep( 100 );

         // Check if any objects were discovered while we were napping.
         discovery_count = 0;
         create_HLA_instance_object_found = false;
         for ( int n = 0; n < obj_count; n++ ) {
            if ( objects[n].is_required() &&
                 objects[n].is_instance_handle_valid() ) {
               discovery_count++;
               if ( objects[n].is_create_HLA_instance() ) {
                  create_HLA_instance_object_found = true;
               }
            }
         }

      // loop while "still missing some objects other than the one for the rejoining
   federate,
      //             or missing some other object(s) but found the rejoining
   federate "
      } while ( ( !create_HLA_instance_object_found &&
                  ( discovery_count < ( required_count - 1 ) ) ) ||
                ( create_HLA_instance_object_found &&
                  ( discovery_count < required_count ) ) );
   }
}
```

Note that the above codes `if` and `while` conditionals use the `create_HLA_instance` flag in their checks of how many objects have been discovered. This is to allow a first time late-joining federate to go through the initialization process, which will allow for the reservation / registration of

the attribute instance object in later sections. This will also allow the execution of the same later sections when the rejoining federate did not resign properly and the instance attributes object was either deleted or made a federation orphan, that object will never be re-discovered.

9.7 Was the instance attribute object discovered?

If the instance attribute object was discovered, by using code similar to below, this means that a rejoining federate was discovered so execute the logic described in Section 9.8. Otherwise, this federate was not a previous member of the federation execution so execute the logic described in Section 9.9.

```
bool MyFederate::is_this_a_rejoining_federate()
{
   // sanity check: see if the object discovery has occurred...
   int discovery_count = 0;
   for ( int n = 0; n < obj_count; n++ ) {
      if ( objects[n].is_instance_handle_valid() ) {
         discovery_count++;
      }
   }

   // have the discovery callbacks occurred already?
   if ( discovery_count > 0) {
      // object discovery has occurred. make sure that the attribute instance
      // object is discovered. if so, then return true.

      // loop thru all object entries...
      for ( int n = 0; n < obj_count; n++ ) {

         // was the required 'create_HLA_instance' object found?
         if ( objects[n].is_required() &&
              objects[n].is_create_HLA_instance() &&
              objects[n].is_instance_handle_valid() ) {

             // set a flag to indicate that this federate is rejoining
             // the federation
             rejoining_federate = true;

            return true;
         } // end of 'if (required & create_HLA_instance & instance_handle_valid )'
      } // end for loop
   }

   return false;
}
```

9.8 The instance attribute object was discovered (federate is rejoining the federation)

If the code we added in Section 9.7 returns true, this federate is rejoining the federation execution. Execute the logic in the following two subsections to regain ownership of the previously divested attributes of the previous execution of this federate.

9.8.1 Re-acquire ownership of all published attributes

In order for the rejoining federate to resume publication of its previously owned attributes, it must regain ownership of all published attributes. This has to occur so that from the first frame of execution, the rejoined federate can begin reflecting its published attributes. Unless this federate regains ownership for all attributes when the federate goes to reflect these attributes, it will receive an '`AttributeNotOwned`' exception on each execution frame for which it does not own the attributes.

This step is accomplished by adding code similar to the following into your federate:

```
for ( int n = 0; n < obj_count; n++ ) {
   if ( objects[n].is_create_HLA_instance() ) {
      objects[n].pull_ownership_upon_rejoin();
   }
}

void MyObject::pull_ownership_upon_rejoin()
{
   // The Set of attribute handle to pull ownership of.
   AttributeHandleSet attr_hdl_set;

   // Lock the ownership mutex since we are processing the ownership pull list.
   ownership_lock();

   // Force the pull ownership of all attributes....
   for ( int i = 0; i < attr_count; i++ ) {

      try {

         // IEEE 1516.1-2000 section 7.18
         if ( attributes[i].is_publish() &&
              attributes[i].is_locally_owned() &&
              !rti_amb->isAttributeOwnedByFederate(
                         obj_instance_handle,
                         attributes[i].get_attribute_handle() ) ) {

            // RTI tells us that the attribute is not owned by this federate. Add
            // attribute handle to the collection for the impending ownership pull.
            attr_hdl_set.insert( attributes[i].get_attribute_handle() );

            // turn off the 'locally_owned' flag on this attribute since the RTI
            // just informed us that we do not own this attribute...
            attributes[i].unmark_locally_owned();
         }

      } catch (ObjectInstanceNotKnown) {
         cout << "MyObject::pull_ownership_upon_rejoin() => "
              << "rti_amb->isAttributeOwnedByFederate() call for published "
              << "attribute '" << attributes[i].FOM_name
              << "' generated an EXCEPTION: ObjectInstanceNotKnown" << endl;
      } catch (AttributeNotDefined) {
         cout << "MyObject::pull_ownership_upon_rejoin() => "
              << "rti_amb->isAttributeOwnedByFederate() call for published "
              << "attribute '" << attributes[i].FOM_name
              << "' generated an EXCEPTION: AttributeNotDefined" << endl;
      } catch (FederateNotExecutionMember) {
         cout << "MyObject::pull_ownership_upon_rejoin() => "
              << "rti_amb->isAttributeOwnedByFederate() call for published "
              << "attribute '" << attributes[i].FOM_name
              << "' generated an EXCEPTION: FederateNotExecutionMember" << endl;
      } catch (SaveInProgress) {
         cout << "MyObject::pull_ownership_upon_rejoin() => "
              << "rti_amb->isAttributeOwnedByFederate() call for published "
              << "attribute '" << attributes[i].FOM_name
              << "' generated an EXCEPTION: SaveInProgress" << endl;
      } catch (RestoreInProgress) {
         cout << "MyObject::pull_ownership_upon_rejoin() => "
              << "rti_amb->isAttributeOwnedByFederate() call for published "
              << "attribute '" << attributes[i].FOM_name
              << "' generated an EXCEPTION: RestoreInProgress" << endl;
      } catch (RTIinternalError & e) {
         wcout << L"MyObject::pull_ownership_upon_rejoin() => "
               << L"rti_amb->isAttributeOwnedByFederate() call for published "
               << L"attribute '" << attributes[i].FOM_name
```

```
            << L"' generated an RTIinternalError: " << e.what() << endl;
      }

   }

   // Make the request only if we do have any attributes for which
   // we need to pull ownership.
   if ( attr_hdl_set.empty() ) {
      cout << "MyObject::pull_ownership_upon_rejoin() No ownership requests were "
           << "added for object '" << name << "'" << endl;
   } else {
      cout << "MyObject::pull_ownership_upon_rejoin() Pulling ownership for "
           << "Attributes of object '" << name << "'" << endl;

      try {
         // IEEE 1516.1-2000 section 7.8
         rti_amb->attributeOwnershipAcquisition(
                        obj_instance_handle,
                        attr_hdl_set,
                        UserSuppliedTag( name, strlen(name)+1) );

      } catch (RTIexception &e) {
         wcout << L"MyObject::pull_ownership_upon_rejoin() => Unable to pull the "
               << L"ownership for the attributes of object '"
               << name << L"' because of error: '"
               << e.what() << L"'" << endl;
      }
   }

   // Unlock the ownership mutex now that we have completed
   // attribute ownership transfer.
   ownership_unlock();
}
```

9.8.2 Wait until ownership transfer is complete

Once the request for the re-acquisition of ownership has occurred, we must wait until all of the RTI callbacks arrive informing us that we have regained ownership of all published attributes. Execute a blocking loop, checking the ownership status of each published attribute with the RTI, until we regain ownership of all published attributes.

This is accomplished by adding code similar to the following into your federate:

```
for ( int n = 0; n < obj_count; n++ ) {
   if ( objects[n].is_create_HLA_instance() ) {
      objects[n].wait_for_ownership_restore_upon_rejoin();
   }
}

void MyObject::wait_for_ownership_restore_upon_rejoin()
{
   // Lock the ownership mutex since we are processing the ownership pull list.
   ownership_lock();

   // count how many attributes we need to wait for in the blocking loop
   int expected_ownership_pulls = 0;
   for ( int i = 0; i < attr_count; i++ ) {

      // look for published attributes only; the locally_owned flag may
      // have been turned off by previous code...
      if ( attributes[i].is_publish() ) {
         expected_ownership_pulls++;
      }
   }
```

```
   // Force the pull ownership of all attributes....
   // perform a blocking loop until ownership of all locally owned
   // published attributes is restored...
   int ownership_counter = 0;
   while ( ownership_counter < expected_ownership_pulls ) {

      // reset ownership count for this loop through all the attributes
      ownership_counter = 0;

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

         try {

            // IEEE 1516.1-2000 section 7.18
            if ( attributes[i].is_publish() &&
                 attributes[i].is_locally_owned() &&
                 rti_amb->isAttributeOwnedByFederate(
                          obj_instance_handle,
                          attributes[i].get_attribute_handle() ) ) {

               ownership_counter++;
            }
         } catch (ObjectInstanceNotKnown) {
            cout << "MyObject::wait_for_ownership_restore_upon_rejoin() => "
                 << "rti_amb->isAttributeOwnedByFederate() call for "
                 << "published attribute '" << attributes[i].FOM_name
                 << "' generated an EXCEPTION: ObjectInstanceNotKnown" << endl;
         } catch (AttributeNotDefined) {
            cout << "MyObject::wait_for_ownership_restore_upon_rejoin() => "
                 << "rti_amb->isAttributeOwnedByFederate() call for "
                 << "published attribute '" << attributes[i].FOM_name
                 << "' generated an EXCEPTION: AttributeNotDefined" << endl;
         } catch (FederateNotExecutionMember) {
            cout << "MyObject::wait_for_ownership_restore_upon_rejoin() => "
                 << "rti_amb->isAttributeOwnedByFederate() call for "
                 << "published attribute '" << attributes[i].FOM_name
                 << "' generated an EXCEPTION: FederateNotExecutionMember" << endl;
         } catch (SaveInProgress) {
            cout << "MyObject::wait_for_ownership_restore_upon_rejoin() => "
                 << "rti_amb->isAttributeOwnedByFederate() call for "
                 << "published attribute '" << attributes[i].FOM_name
                 << "' generated an EXCEPTION: SaveInProgress" << endl;
         } catch (RestoreInProgress) {
            cout << "MyObject::wait_for_ownership_restore_upon_rejoin() => "
                 << "rti_amb->isAttributeOwnedByFederate() call for "
                 << "published attribute '" << attributes[i].FOM_name
                 << "' generated an EXCEPTION: RestoreInProgress" << endl;
         } catch (RTIinternalError & e) {
            wcout << L"MyObject::wait_for_ownership_restore_upon_rejoin() => "
                  << L"rti_amb->isAttributeOwnedByFederate() call for "
                  << L"published attribute '" << attributes[i].FOM_name
                  << L"' generated an RTIinternalError: " << e.what() << endl;
         }
      } // end of 'for' loop

      usleep( 100 ); // sleep for a bit as not to hammer the RTI with requests...
   } // end of 'while' loop

   // Unlock the ownership mutex now that we have completed attribute
   // ownership transfer.
   ownership_unlock();
}
```

Now, skip over the next section, since it does not pertain to rejoining a federation, and go to Section 9.10.

9.9 The instance attribute object was not discovered (federate is joining the federation for the first time)

If code we added in Section 9.7 returned false, the attribute instance object was not discovered in the federation so we need to proceed with joining the federation run for the first time.

9.9.1 Reserve Object Instance Names

The late-joiner federate next reserves all the relevant object instance names, **excluding** the already registered "SimConfig" name, following the procedures outlined in section 6.4 above. The names to be used in a federation should be agreed upon in advance.

9.9.2 Wait for All Object Instance Name Reservations

The late-joiner federate next waits for all the object instance name reservation callbacks following the procedure outlined in section 6.5 above, but with a reserved status flag specific to each object instance name.

9.9.3 Register Object Instances

Now that the late-joiner federate has published all objects, it needs to create instances of the objects to update **excluding** the *Simulation Configuration* object. The procedures in section 6.6 above are followed with the exception that an instance for the *Simulation Configuration* object is not created.

9.10 Wait for All Required Objects to be Registered

Now that the late-joiner federate has created the object instances it will be updating (section 9.9) or found the object instance that was previously created (section 9.7), it needs to wait for all the required object instances it will be using to be registered by following procedures in section 6.7 above. The object instances the late-joiner federate uses include those it will update, the Simulation Configuration, and those it will receive RTI reflections for from the other federates. Unless a federate is required, we must not wait for its corresponding data object instance to show up because it will not start being reflected until shortly after the non-required federate joins the federation.

If the federate is rejoining the federation and its attribute instance object has been discovered, this section is really a sanity check to ensure that all of the required objects have joined the federation.

If any of the object's name reservation fails with the RTI from section 9.9.1 above, the RTI will trigger the `FedAmbassador::objectInstanceNameReservationFailed()` callback. It is the federate's duty to report this failure with appropriate error message(s) to the user and terminate the federate because we experienced an unrecoverable error.

One of the reasons why an object name reservation would fail is that this object name has already been reserved and per standards an object name can only be reserved once during the lifetime of the federation (see Sections 6.2 and 6.3 of [1]). This scenario occurs because a previously joined federate reserved its object name with the RTI and that named object either no longer exists (was deleted from the federation) or is a federation orphan (not discoverable). This is exactly what the user would see if the rejoining federate did not resign properly. See Section 16 for the proper way to resign a federate so it can rejoin the federation execution at a later time.

Add code like the following to your FedAmbassador subclass and Federate to print the reason why the federate name reservation failed:

```
void MyFedAmb::objectInstanceNameReservationFailed(wstring const &
   theObjectInstanceName)
```

```
throw (UnknownName,
       FederateInternalError)
{
   if ( my_fed != NULL ) {
         wcout << L"MyFedAmb::objectInstanceNameReservationFailed() FAILED '"
               << theObjectInstanceName.c_str() << L"'";

      my_fed->object_instance_name_reservation_failed( theObjectInstanceName );
   }
}

void MyFederate::object_instance_name_reservation_failed(
   wstring const & obj_instance_name )
{
   // For multiphase initialization version 1, we handle the SimConfig
   // instance name reservation failure to help determine the master.
   if ( ( multiphase_initialization_version == 1 ) &&
        ( obj_instance_name == L"SimConfig" ) ) {

      // We are NOT the Master federate since we failed to reserve the
      // "SimConfig" name.
      set_master( false );

      // We failed to register the "SimConfig" name which means that another
      // federate has already registered it.
      sim_config.set_name_registered();

      cout << "MyFederate::object_instance_name_reservation_failed() 'SimConfig'"
           << endl;
   } else {

      wcout << L"MyFederate::object_instance_name_reservation_failed() Name:'"
            << obj_instance_name.c_str()
            << L"' Please check to make sure the object instance name is unique, "
            << L"no duplicates, within the Federation. For example, try using "
            << L"fed_name.object_FOM_name for the object instance name. Also, an "
            << L"object should be owned by only one Federate so one common "
            << L"mistake is to have the 'create_HLA_instance' flag for the same "
            << L"object being set to true in more than one Federate."
            << endl;

      wstring obj_name;
      string str;
      for ( int n = 0; n < obj_count; n++ ) {
         str = objects[n].get_name();               // get object name
         obj_name.assign( str.begin(), str.end() ); // make a wstring for compare
         if ( obj_name == obj_instance_name ) {
            if ( objects[n].is_create_HLA_instance() ) {
               cout << "   ** You specified that this Federate can rejoin "
                    << "the Federation but the original instance attributes "
                    << "could not be located in order to re-acquire ownership. "
                    << "They were either deleted, or are orphans in the "
                    << "Federation with no possibility of regaining ownership. **"
                    << endl
                    << "   ** In order for the rejoin to succeed, you must "
                    << "resign this Federate with the directive to divest "
                    << "ownership of its instance attributes. See Section 16 of "
                    << "\"IMSim_Multiphase_Init_Design_Document\". **"
                    << endl
                    << "   ** Note: In order for the Federation rejoin to be "
                    << "successful, make sure that there is at least one other "
                    << "Federate set up to publish at least one of the "
                    << "attributes (by setting the 'publish' flag to true in "
                    << "another Federate). This is necessary for the successful "
                    << "transfer of ownership which keeps the instance "
                    << "attribute's object from becoming a Federation "
```

```
                    << "orphan. **"
                    << endl;
            } // end of has create HLA instance flag
         } // end of name match
      } // end of for loop thru objects

      // Bad things have happened if the name reservation failed since it should
      // be unique in the federation, so quit the simulation.
      exit(1);
   }
}
```

9.11 Request the number of and the names of all joined federates from the MOM

See section 8.8 for these procedures.

9.12 Is HLA Time Management Being Used?

If HLA time management is being used then proceed to step 9.13. If HLA time management is not being used then Central Timing Equipment (CTE) shall be used to synchronize federates by way of a common clock. A late-joining federate not using HLA time-management but using CTE will have to wait for a go-to-run interaction message that specifies the CTE time the federate will go to run at. Skip to step 9.16 once the federate modes to run.

9.13 Setup Time Management

Follow the procedures outlined in section 8.9 to setup time management.

9.14 Query the GALT

A late joining federate will have an HLA logical time that is not synchronized with the logical time of the other running federates. We can query the RTI ambassador to get the Greatest Allowable Logical Time (GALT), which will allow the late joining federate to catch-up to the logical time of the other federates. The GALT for a federate will be undefined if there are no other time regulating federates because the federate can advance to any logical time without bounds.

```
auto_ptr<LogicalTime> galt = RTI_amb->queryGALT();
```

9.15 Time Advance Request to the GALT

Make a time advance request to the GALT to allow the late joining federate to catch-up to the logical of the other running federates.

```
RTI_amb->timeAdvanceRequest( galt );
```

9.16 Join the simulation execution, already in progress

Once the retrieval of the name of all joined federates has been completed, the late joining federate is ready to join the simulation execution, which is already in progress (see section 10).

10 Federate Execution

Once the required federates have cleared the above Master (see section 6), Non-Master (see section 7) and Late-Joiner (section 9) Simulation Configurations as well as being synchronized on the "startup_v2" sync-point (see sections 12.15 and 13.8), the federation is ready to execute. While the

federation is executing, any other federates which join the federation automatically execute the "Late-Joiner" leg of the multiphase initialization (see section 9).

11 How to configure the Pitch RTI for federation save and restore

The user must specify where the Pitch RTI is to store / reload it's save files for the CRC and the LRC. The location is specified in the "CRC.save-path" value in the "pRTI516CRC.settings" & "LRC.save-path" value in the "pRTI1516LRC.settings" files found in the 'prti1516' directory. Both of these can point to the same directory but that directory must exist prior to the RTI attempting to save into it.

12 How to restore a saved federation (the first federate in the federation)

A federation can be restored from a checkpoint save set only by the first federate which joins the federation, thereby creating the federation. The first federate must perform checks to ensure that only federates which were executing when the federation was saved are allowed to rejoin the restored federation. After all previously running federates have joined, it coordinates the rest of the federation restore until the federation is ready to resume execution.

This procedure is represented in the "Load a Check-point?" leg on the far left branch of the Multiphase Initialization Process Flowchart in Appendix B. The simulation designer is responsible for adapting their simulation to follow the restore portion of the multiphase initialization scheme, covered in this chapter, to initiate the restore of the federation.

To ensure that only previously-joined federates rejoin the federation, a shortcut was developed. The names of all federates running at the time of the federation save, and their corresponding MOM names, were saved into a separate file (see section 15.8). The motivation for doing this was to speed up the federation startup by quickly loading the required federate information and waiting for the required federates to join the federation without loading the first federate's checkpoint file to get to this information.

In this chapter, I am using an `enum` to describe the state of the restore activity, called `restore_process`, and its pre-restore counterpart, `pre_restore_process`; we need to keep a copy of this value since the original one will be lost when the checkpoint file is loaded into memory. The `enum` contains these possible (self-explanatory) states: `No_Restore`, `Restore_Request_Failed`, `Restore_Request_Succeeded`, `Initiate_Restore`, `Restore_In_Progress`, `Restore_Complete`, `Restore_Failed`. These states will be used throughout the code to determine, at any time, the current state of the restore so we can take appropriate actions. Additionally, we make use of other flags, which hopefully will be self-explanatory.

The federation restore is broken down into these sub-sections:

12.1 Read the '.running_feds' file and update list of "Required" federates
12.2 Wait for the required federates to join the federation
12.3 Look for any "unauthorized" federates
12.4 Load the checkpoint file
12.5 Request federation restore
12.6 Wait for restore request callback
12.7 Wait for the "restore begun" callback
12.8 Wait for the "ready to restore" callback

12.9 Inform RTI of the success/failure of the federate restore
12.10 Wait until the federation is restored
12.11 Register "startup_v2" sync point with all joined federates
12.12 Wait for the announcement of "startup_v2" sync point
12.13 Restart the federate (including setting up all RTI handles)
12.14 Achieve and wait for the "startup_v2" sync point
12.15 Resume execution of the federation

12.1 Read the '.running_feds' file and update list of "Required" federates

Read the '`.running_feds`' file, whose creation and content are described in section 15.8, and replace the contents of the data structure which is used to determine the federates required for federation startup with the data read from the file. This is a necessary step because we need to get the federation restored back into the identical state it was in when it was saved and requiring only the same federates to re-join is the first step toward achieving this goal.

12.2 Wait for the required federates to join the federation

Follow the procedures in section 6.1. This step will wait indefinitely until all previously joined federates have joined. However, since we are tightly bound to which federates can join at startup, we now need to add logic to recognize and keep a list of any "unauthorized" federates in the `MyFederate::set_federate_instance_name()` routine discussed in section 6.1.

12.3 Look for any "unauthorized" federates

Checking for "unauthorized" federates must be done to preserve the state of the reloaded federation. The federation cannot be restored to a previously run state when extra federate(s) join the federation before the federation has completed the restore and begins executing. The RTI would recognize this condition and would fail the restore. However, the RTI would not provide us a reason why it failed the restore, much less provide us a list of "unauthorized" federate(s), which is useful to know if that was the reason why the federation failed to restore.

If any federate joined the federation which was not running when the federation was saved (i.e., not found in the required federates list loaded in section 12.1), the master federate must report this by printing an appropriate message listing the "unauthorized" federate(s) as well as any other pertinent information, and terminate the federation.

12.4 Load the checkpoint file

Load the checkpoint file, replacing every saved data structure in memory with the contents read from the checkpoint file.

It is beyond the scope of this document to tell the user how to do this.

12.5 Request federation restore

Once all of the required federates have joined, we need to request the federation restore from the RTI, supplying the `checkpoint_set_name` in the call.

```
RTIAmbassador->requestFederationRestore( checkpoint_set_name );
```

12.6 Wait for restore request callback

The RTI will send callbacks whether the request succeeded (`MyFedAmbassador::requestFederationRestoreSucceeded()`) or failed

(`MyFedAmbassador::requestFederationRestoreFailed()`). These routines should update the state of the `restore_process enum`, as show below:

```
void MyFedAmbassador::requestFederationRestoreSucceeded(wstring const & label)
   throw (FederateInternalError) = 0;
{
   myfederate->set_restore_process( Restore_Request_Successed );
}

void MyFedAmbassador::requestFederationRestoreFailed(wstring const & label)
   throw (FederateInternalError)
{
   myfederate->set_restore_process( Restore_Request_Failed );
}
```

In our federate, we need to go into a wait loop until we receive one of the two `enum` state updates.

```
void MyFederate::wait_for_restore_request_callback()
{
   while ( ( restore_process != Restore_Request_Failed ) &&
           ( restore_process != Restore_Request_Succeeded ) ) {
      usleep ( 100 ); // sleep until RTI responds...
   }
}
```

If we receive a failure, we terminate the federation, making sure to print a meaning full error message.

```
// wait for the success / failure response from the RTI
myfederate->wait_for_restore_request_callback();

if (myfederate->has_restore_request_failed() ) {
   fatalError("FATAL ERROR: RTI rejected the  restore request!!!! "
              "Make sure that you are restoring the federates from an identical"
              " federation save set.\n      See IEEE 1516.1-2000, Section 4.18 for"
              " further info for the reasons why the RTI would reject the "
              "federation restore request...";
}
```

12.7 Wait for the "restore begun" callback

When the callback `MyFedAmbassador::requestFederationRestoreSucceeded()` routine is triggered for all federates, the RTI then sends out two signals to all federates when it is ready to proceed with the restore: that the restore is imminent (discussed here), and that the RTI is ready for all the federates to restore themselves (discussed in the next section). These are, in essence, two internal pause points that all federates must receive before they can safely restore themselves. To accomplish this, a flag would be set by `MyFedAmbassador` to inform the federate that the federation is in 'restore begun' mode.

```
void MyFedAmbassador::federationRestoreBegun()
throw ( RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->set_restore_begun();
}
```

Then, in our code, we would execute a wait loop waiting until the MyFedAmbassador has set this flag.

```
void MyFederate::wait_for_federation_restore_begun()
{
   while ( ! restore_begun ) {
      usleep( 1000 ); // sleep until RTI responds...
   }
}
```

12.8 Wait for the "ready to restore" callback

The RTI informs us, via the `MyFedAmbassador::initiateFederateRestore()` callback, that the federation is ready for the federate to restore itself, using the provided `label`. We would set the '`start_to_restore`' flag and save the provided `label` into our federate as its `restore_name`.

```
void MyFedAmbassador::initiateFederateRestore(
       wstring  const & label,
       FederateHandle const & handle)
throw ( RTI1516_NAMESPACE::SpecifiedSaveLabelDoesNotExist,
        RTI1516_NAMESPACE::CouldNotInitiateRestore,
        RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->set_restore_name( label );
   myfederate->set_start_to_restore( true );
}
```

The federate would execute a wait loop until the '`start_to_restore`' flag has been set by MyFedAmbassador.

```
void MyFederate::wait_until_federation_is_ready_to_restore()
{
   while ( ! start_to_restore ) {
      usleep( 1000 ); // sleep until RTI responds...
   }
}
```

12.9 Inform RTI of the success/failure of the federate restore

We need to inform the RTI of the success or failure of the checkpoint file read. Since we already loaded our checkpoint in section 12.4, it is just a matter of checking our state and making the appropriate RTI call.

```
if ( prev_restore_process == Restore_Complete ) {
   try {
      RTIAmbassador->federateRestoreComplete();
   } catch ( RestoreNotRequested ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore complete"
           << " -- EXCEPTION: RestoreNotRequested" << endl;
   } catch ( FederateNotExecutionMember ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore complete"
           << " -- EXCEPTION: FederateNotExecutionMember" << endl;
   } catch ( SaveInProgress ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore complete"
           << " -- EXCEPTION: SaveInProgress" << endl;
   } catch ( RTIinternalError &e ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore complete"
           << " -- EXCEPTION: RTIinternalError: " << e.what() << endl;
   }
} else if ( prev_restore_process == Restore_Failed ) {
   try {
      RTIAmbassador->federateRestoreNotComplete();
   } catch ( RestoreNotRequested ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore NOT "
           << "complete -- EXCEPTION: RestoreNotRequested" << endl;
   } catch ( FederateNotExecutionMember ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore NOT "
           << "complete -- EXCEPTION: FederateNotExecutionMember" << endl;
   } catch ( SaveInProgress ) {
      cout << "Federate::inform_RTI_of_restore_completion() -- restore NOT "
           << "complete -- EXCEPTION: SaveInProgress" << endl;
   } catch ( RTIinternalError &e ) {
```

```
      cout << "Federate::inform_RTI_of_restore_completion() -- restore NOT "
           << "complete -- EXCEPTION: RTIinternalError: " << e.what() << endl;
   }
}
```

12.10 Wait until the federation is restored

Now that we informed the RTI of our restore state, we must wait until the rest of the federates complete their own restore. We must be diligent to perform error checking at every step of this waiting process since any federate restore can fail; we must identify and report any errors.

We should check the state of our restore, as outlined in the code below, and if any problems are found, return a string that will be used to print an appropriate error message when terminating the federation. If no problems were found, go into a wait loop until we receive a callback from the RTI that the federation restore has completed.

If, in the middle of the wait loop, a federate resigns, generate an appropriate error message that will be printed upon federation termination and exit the routine.

If all went well with all federate restores, exit with an empty string from the routine to signal that the federation restore was successful. Otherwise, compose and return an appropriate error message.

```
string MyFederate::wait_for_federation_restore_to_complete()
{
   string tRetString;

   if ( restore_failed ) {
      tRetString = "MyFederate::wait_for_federation_restore_to_complete() => "
                   "restore of federate failed\nTERMINATING SIMULATION!";
      return tRetString;
   }

   if ( federation_restore_failed_callback_complete ) {
      tRetString = "mYFederate::wait_for_federation_restore_to_complete() => "
                   "federation restore failed\nTERMINATING SIMULATION!";
      return tRetString;
   }

   if ( restore_process == Restore_Failed ) {
      // before we enter the wait loop, the RTI informed us that it accepted
      // the failure of the federate restore. build and return a message.
      tRetString = "MyFederate::wait_for_federation_restore_to_complete() => "
                   "federation restore FAILED! Look at the message from the "
                   "Federate::print_restore_failure_reason() routine "
                   "for a reason why the federation restore failed.\n"
                   "TERMINATING SIMULATION!";
      return tRetString;
   }

   // nobody reported any problems, wait until the restore is completed.
   while ( ! restore_completed ) {
      if ( running_feds_count_at_time_of_restore > running_feds_count ) {
         // someone has resigned since the federation restore has been initiated.
         // build a message detailing what happened and exit the routine.
         tRetString = "MyFederate::wait_for_federation_restore_to_complete() => "
                      "while waiting for restore of the federation => "
                      "a federate resigned before the federation restore "
                      "completed!\nTERMINATING SIMULATION!";
         return tRetString;
      } else {
         usleep( 100 ); // sleep until RTI responds...
      }
   }
```

```
   if ( restore_process == Restore_Failed ) {
      // after this federate restore wait loop has finished, check if the
      // RTI accepted the failure of the federate restore. build and return a
      // message.
      tRetString = "MyFederate::wait_for_federation_restore_to_complete() => "
                   "federation restore FAILED! Look at the message from the "
                   "MyFederate::print_restore_failure_reason() routine "
                   "for a reason why the federation restore failed.\n"
                   "TERMINATING SIMULATION!";
      return tRetString;
   }

   return tRetString;
}
```

Call the above routine.

```
string tStr = myfederate->wait_for_federation_restore_to_complete();
if ( tStr.length() ) {
   myfederate->wait_for_federation_restore_failed_callback_to_complete();
   fatalError( "FATAL ERROR: you indicated that you wanted to restore a check"
               "point => wait_for_federation_restore_to_complete() failed "
               "with this message '%s'", tStr.c_str() );
}
```

If it returns an error message (i.e., not an empty string), wait until the RTI notifies every federate that the restore failed. Once we receive the callback that the federation restore failed, we print out the contents of the returned error message and terminate the federation.

```
void MyFederate::wait_for_federation_restore_failed_callback_to_complete()
{
   while ( ! federation_restore_failed_callback_complete ) {
      // if the federate has already been restored, do not wait for a signal
      // from the RTI that the federation restore failed, you'll never get it!
      if (restore_completed ) {
         return;
      }
      usleep ( 100 ); // sleep until RTI responds...
   }
}
```

If any federate(s) failed to restore successfully, all federates receive a RTI callback to the `MyFedAmbassador::federationNotRestored()` method, sending along a `RestoreFailureReason` object containing the reason(s) for failure. We would need to decode the supplied parameter to determine and report the cause(s) of the restore failure.

```
void MyFedAmbassador::federationNotRestored (
   RestoreFailureReason theRestoreFailureReason)
throw ( RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->set_restore_failed();
   myfederate->print_restore_failure_reason( theRestoreFailureReason );
}
```

The `FedAmbassador` code would call the following code snippet to decode and print the failure reason(s), set a flag that we received the restore failed callback from the RTI and gracefully exit the simulation.

```
void MyFederate::print_restore_failure_reason( RestoreFailureReason theReason )
{
   ostringstream errmsg;
```

```
   if (theReason == RTI_UNABLE_TO_RESTORE ) {
      errmsg << "MyFederate::print_restore_failure_reason(): reason="
             << "\"RTI_UNABLE_TO_RESTORE\"" << endl;
   }
   if (theReason == FEDERATE_REPORTED_FAILURE_DURING_RESTORE ) {
      errmsg << "MyFederate::print_restore_failure_reason(): reason="
             << "\"FEDERATE_REPORTED_FAILURE_DURING_RESTORE\"" << endl;
   }
   if (theReason == FEDERATE_RESIGNED_DURING_RESTORE ) {
      errmsg << "MyFederate::print_restore_failure_reason(): reason="
             << "\"FEDERATE_RESIGNED_DURING_RESTORE\"" << endl;
   }
   if (theReason == RTI_DETECTED_FAILURE_DURING_RESTORE ) {
      errmsg << "MyFederate::print_restore_failure_reason(): reason="
             << "\"RTI_DETECTED_FAILURE_DURING_RESTORE\"" << endl;
   }

   federation_restore_failed_callback_complete = true;

   // gracefully exit the simulation.
   fatalError( errmsg );
}
```

12.11 Request the federate handles from RTI

Since all federates and the RTI were just restored to a previously running state, there is no guarantee that the user restarted the federation in the identical order as the saved federation. This could lead up into an inconsistent state when it comes to the contents of the "joined_federate_set" which is used to register sync points with an exact set of federates.

We already ensure that only the federates which were running were allowed, by federate name, to rejoin the federation. In the case when one or more federates resign from the federation before the federation is saved, the federate handles between the current and reloaded federations would not match.

What we have in memory are handles for federates as they joined the federation before the federation was restored. Because RTI handles are immutable (as discussed in section 15.9), they cannot be checkpointed. Therefore, they must be retrieved from the RTI after a federation restore and before proceeding with restart of the reloaded federate.

Call the following code to retrieve the federate handles from the MOM:

```
void MyFederate::ask_MOM_for_restored_federate_handles()
{
   // make sure that we are in federate handle rebuild mode...
   fed_amb->set_federation_restored__rebuild_federate_handle_set();

   // clear the federate handle set
   joined_federate_set.clear();


   // Use the MOM to get the list of registered federates.
   this->MOM_federate_class_handle = RTI_amb->getObjectClassHandle(
             L"HLAobjectRoot.HLAmanager.HLAfederate" );
   this->MOM_federate_handle = RTI_amb->getAttributeHandle(
             MOM_federate_class_handle, L"HLAfederateHandle" );

   AttributeHandleSet fedMomAttributes;
   fedMomAttributes.insert( MOM_federate_handle );

   // Subscribe the MOM object attributes.
   RTI_amb->subscribeObjectClassAttributes( MOM_federate_class_handle,
                                            fedMomAttributes,
```

```
                                            true );

   AttributeHandleSet requestedAttributes;
   requestedAttributes.insert( MOM_federate_handle );

   // Request initial values.
   RTI_amb->requestAttributeValueUpdate( MOM_federate_class_handle,
                                         requestedAttributes,
                                         UserSuppliedTag( 0, 0 ) );

   // wait until all of the federate handles have been retrieved.
   while( joined_federate_set.size() != (unsigned int) running_feds_count ) {
      usleep( 1000 );
   }


   // We are done with the MOM interface to unsubscribe from the classes we used.
   RTI_amb->unsubscribeObjectClass( MOM_federate_class_handle );

   // make sure that we are no longer in federate handle rebuild mode...
   fed_amb->reset_federation_restored__rebuild_federate_handle_set();
}
```

The federate handles would be sent to us via the `MyFedAmbassador::reflectAttributeValues()` callback. In this routine, it is necessary to add checks if the objectClass matches the `MOM_federation_class` we received from the RTI above and if the fed ambassador's `federation_restored__rebuild_federate_handle_set` boolean has been set to decode the federate handles and adding them to federate handle set by calling the following routine:

```
void MyFederate::rebuild_federate_handle(
   ObjectInstanceHandle                id,
   auto_ptr< AttributeHandleValueMap > values )
{
   AttributeHandleValueMap::iterator attr_iter;

   // loop through all federate handles
   for( attr_iter = values->begin(); attr_iter != values->end(); attr_iter++ ) {

      int num_bytes = attr_iter->second.size();
      int * data = (int *)attr_iter->second.data();

      // Do a sanity check on the size.
      if ( num_bytes != 8 ) {
         cout << "MyFederate::rebuild_federate_handles():"
              << __LINE__ << " Unexpected number of bytes in the"
              << " FederateHandle because the byte count is " << num_bytes
              << " but we expected 8!\nTERMINATING FEDERATE";
         exit(1);
      }


      // The HLAfederateHandle has the HLAhandle datatype which is has the
      // HLAvariableArray encoding with an HLAbyte element type.
      //  0 0 0 4 0 0 0 2
      //  ---+--- | | | |
      //     |     ---+---
      // #elem=4  fedID = 2
      //
      // Determine if we need to byteswap or not since the FederateHandle
      // is in Big Endian. First 4 bytes (first 32-bit integer) is the number
      // of elements.
      int fed_id = is_transmission_byteswap( THLA_BIG_ENDIAN ) ?
                      byteswap_int( data[1] ) : data[1];
```

```
      // Add this FederateHandle to the set of joined federates.
      joined_federate_set.insert( FederateHandle( fed_id ) );
   }
}
```

12.12 Register "startup_v2"sync point with all joined federates

Now that the first federate restore is complete and the federate handle set has been restored to what it was when the federation was running last, in order to ensure that we do not introduce a race condition by allowing this federate to run prematurely before all federates have finished their restore, we must register "startup_v2" synchronization point with all joined federates. The following call creates this sync point.

```
rtiAmbassador->registerFederationSynchronizationPoint( L"startup_v2",
                  UserSuppliedTag( 0, 0 ), joined_federate_set );
```

If the sync point registration is successful, we will receive the callback from the RTI, via the FedAmbassador::synchronizationPointRegistrationSucceeded() method, detailed in section 6.8. If the sync point registration fails, we will receive the callback from the RTI, via the FedAmbassador::synchronizationPointRegistrationFailed() method, also detailed in section 6.8.

12.13 Wait for the announcement of "startup_v2"sync point

To avoid a race condition of the RTI not delivering the "startup_v2" sync point announcement before we attempt to synchronize on it (in section 12.15), we must execute a blocking loop waiting for the RTI to deliver the announcement before proceeding.

12.14 Restart the federate (including setting up all RTI handles)

The IEEE_1516.1-2000, section 1.4.3 defines a RTI handle as "capable of being manipulated by a computer or for communication between a federate and the RTI. Originated by the RTI, federation execution-wide unique, and unpredictable." [1]. Thus, they are not checkpoint-able because their value is "unpredictable".

What was saved in the checkpoint file is enough unique information with which we can request the handle for each element back from the RTI in order to restore the federation back into its previously running state.

This sub-section requires that the following are restored and re-registered with the RTI (unless otherwise noted):

12.14.1 Restore sync points

Restore any synchronization points from the checkpoint file into memory. However, do not re-register them with the RTI since they would be restored into the RTI when it reloads its checkpoint, thus avoiding triggering duplicate sync point registration exceptions.

12.14.2 Restore pending interactions

Restore any pending interactions from the checkpoint file and re-discover the ownership object from the RTI. This will bring the interaction back to their pre-checkpoint state so they can take place in this execution frame.

12.14.3 Restore objects and their attributes

Each federate has to restore RTI handles for each object and each one of its attributes. This is accomplished by executing the following code:

```
   const char * obj_FOM_name = "";
   const char * attr_FOM_name = "";
   string tmpStr;
   wstring ws_FOM_name = L"";
   int FOM_name_type = 0; //0:N/A 1:Object 2:Attribute  What name type we have

   //--------------------------------------------------------
   // Initialize the Object and Attribute RTI handles.
   //--------------------------------------------------------
   try {

      // Resolve all the handles/ID's for the objects and attributes.
      for ( int n = 0; n <  data_obj_count; n++ ) {
         // Create the wide-string object FOM name.
         FOM_name_type = 0; // NA
         obj_FOM_name = data_objects[n].get_FOM_name();
         FOM_name_type = 1; // Object
         tmpStr = obj_FOM_name;
         ws_FOM_name.assign( tmpStr.begin(), tmpStr.end() );

         // Get the class handle for the given object FOM name.
         data_objects[n].set_class_handle(
            RTIAmbassador->getObjectClassHandle( ws_FOM_name ) );

         int attr_count = data_objects[n].get_attribute_count();
         Attribute * attrs = data_objects[n].get_attributes();

         // Resolve the handles/ID's for the attributes.
         for ( int i = 0; i < attr_count; i++ ) {

            // Create the wide-string Attribute FOM name.
            FOM_name_type = 0; // N/A
            attr_FOM_name = attrs[i].get_FOM_name();
            FOM_name_type = 2; // Attribute
            tmpStr =  attr_FOM_name;
            ws_FOM_name.assign( tmpStr.begin(), tmpStr.end() );

            // Get the Attribute-Handle from the RTI.
            attrs[i].set_attribute_handle(
               RTIAmbassador->getAttributeHandle(
                              data_objects[n].get_class_handle(),
                              ws_FOM_name ) );
         }
      }
   } catch ( NameNotFound &e ) {
      switch ( FOM_name_type ) {
         case 1: // Object
            fatalError( "setup_object_RTI_handles() Object FOM Name '%s' Not "
                        "Found. Please double check your inputs to make sure the"
                        " Object FOM Name is correctly specified.",
                        obj_FOM_name );
            break;
         case 2: // Attribute
            fatalError( "setup_object_RTI_handles() For Object FOM Name '%s'"
                        ", Attribute FOM Name '%s' Not Found. Please double "
                        "check your inputs to make sure the Object Attribute"
                        " FOM Name is correctly specified.", obj_FOM_name,
                        attr_FOM_name );
            break;
         default: // FOM name we are working with is unknown.
            fatalError( "setup_object_RTI_handles() Object or Attribute FOM "
                        "Name Not Found. Please double check your inputs to "
                        "make sure the FOM Name is correctly specified." );
            break;
      }
   } catch ( FederateNotExecutionMember ) {
```

```
      fatalError("setup_object_RTI_handles() Federate Not Execution Member" );
   } catch ( RTIinternalError &e ) {
      fatalError ( "setup_object_RTI_handles() RTIinternalError: '%ls'",
                   e.what());
   } catch ( RTI1516_EXCEPTION &e ) {
      fatalError ("setup_object_RTI_handles() RTI1516_EXCEPTION for '%ls'",
                   e.what() );
   }
```

12.14.4 Interactions and their parameters

Each federate has to restore RTI handles for each interaction and each one of its parameters. This is accomplished by executing the following code:

```
   const char * inter_FOM_name = "";
   const char * param_FOM_name = "";
   string tmpStr;
   wstring ws_FOM_name = L"";
   int FOM_name_type = 0; //0:NA 1:Interaction 2:Parameter  What name we are dealing
   with.

   //-----------------------------------------------------
   // Initialize the Interaction and Parameter RTI handles.
   //-----------------------------------------------------
   try {

      // Process all the Interactions.
      for ( int n = 0; n < inter_count; n++ ) {

         // The Interaction FOM name.
         FOM_name_type = 0; // NA
         inter_FOM_name = interactions[n].get_FOM_name();
         FOM_name_type = 1; // Interaction
         tmpStr = inter_FOM_name;
         ws_FOM_name.assign( tmpStr.begin(), tmpStr.end() );

         // Get the Interaction class handle.
         interactions[n].set_class_handle(
            RTIAmbassador->getInteractionClassHandle( ws_FOM_name ) );

         // The parameters.
         int param_count = interactions[n].get_parameter_count();
         Parameter * params = interactions[n].get_parameters();

         // Process the parameters for the interaction.
         for ( int i = 0; i < param_count; i++ ) {

            // The Parameter FOM name.
            FOM_name_type = 0; // NA
            param_FOM_name = params[i].get_FOM_name();
            FOM_name_type = 2; // Parameter
            tmpStr = param_FOM_name;
            ws_FOM_name.assign( tmpStr.begin(), tmpStr.end() );

            // Get the Parameter Handle.
            params[i].set_parameter_handle(
                      RTIAmbassador->getParameterHandle(
                                  interactions[n].get_class_handle(),
                                  ws_FOM_name ) );
      }
   } catch ( NameNotFound &e ) {
      switch ( FOM_name_type ) {
         case 1: // Interaction
            fatalError( "setup_interaction_RTI_handles() Interaction FOM Name "
                        "'%s' Not Found. Please double check your inputs to make "
```

```
                    " sure the Interaction FOM Name is correctly specified.",
                    inter_FOM_name );
            break;
         case 2: // Parameter
            fatalError( "setup_interaction_RTI_handles() For Interaction FOM "
                        "Name '%s', Parameter FOM Name '%s' Not Found. Please"
                        " double check your inputs to make sure the Interaction"
                        " Parameter FOM Name is correctly specified.",
                        inter_FOM_name, param_FOM_name );
            break;
         default: // FOM name we are working with is unknown.
            fatalError( "setup_interaction_RTI_handles() Interaction or Parameter"
                        " FOM Name Not Found. Please double check your inputs to "
                        "make sure the FOM Name is correctly specified." );
            break;
      }
   } catch ( FederateNotExecutionMember ) {
      fatalError( "setup_interaction_RTI_handles() FederateNotExecutionMember " );
   } catch ( RTIinternalError &e ) {
      fatalError( "setup_interaction_RTI_handles() RTIinternalError: '%ls'.",
                  e.what() );
   } catch ( RTI1516_EXCEPTION &e ) {
      fatalError( "setup_interaction_RTI_handles() exception for '%ls'.",
                  e.what() );
   }
```

12.14.5 Any pending ownership transfer commands

Re-encode the data read from the checkpoint file back into the ownership transfer object in memory so that they can occur in this execution frame.

12.15 Achieve and wait for the "startup_v2" sync point

See sections 8.10 & 8.11 for these procedures.

12.16 Resume execution of the federation

See section 10 for this procedure.

13 How to restore a saved federation (not the first federate in the federation)

A federation restore can be initiated by any federate. However, since we are restoring the federation when starting up the simulation, it only makes sense that the federation restore gets initiated by the federate that created the federation, which is **not** this federate. Any subsequent requests to restore a federation after the first federate already made the request, would receive a "Restore in progress" non-fatal exception.

Since the federate who created the federation will take care of RTI communications, all that this, the "non-master" federate, has to do is listen for the RTI callbacks to inform it when to restore itself, load the checkpoint file, restart the federate and get itself ready to execute.

This procedure is represented in the "Load a Check-point?" leg on the second from the right branch of the flowchart in Appendix B.

13.1 Load the checkpoint file
13.2 Wait for the "restore begun" callback
13.3 Wait for the "ready to restore" callback
13.4 Inform RTI of the success/failure of the federate restore

13.5 Wait until the federation is restored
13.6 Wait for the announcement of "startup_v2"sync point
13.7 Restart the federate (including setting up all RTI handles)
13.8 Achieve and wait for the "startup_v2"sync point
13.9 Resume execution of the federation

13.1 Load the checkpoint file

See section 12.4 for this procedure.

13.2 Wait for the "restore begun" callback

See section 12.7 for this procedure.

13.3 Wait for the "ready to restore" callback

See section 12.8 for this procedure.

13.4 Inform RTI of the success/failure of the federate restore

See section 12.9 for this procedure.

13.5 Wait until the federation is restored

See section 12.10 for this procedure.

13.6 Wait for the announcement of "startup_v2"sync point

See section 12.13 for this procedure.

13.7 Restart the federate (including setting up all RTI handles)

See section 12.14 for this procedure.

13.8 Achieve and wait for the "startup_v2"sync point

See sections 8.10 & 8.11 for these procedures.

13.9 Resume execution of the federation

See section 10 for this procedure.

14 Initiating a Federation Save

It is recommended that the federation be programmed to request a single federation save at a time. It is up to the designer to construct their federation so it follows this guideline so as not to deluge the RTI with multiple federation save requests for the same exact point in time. The request for a federation save can come from any federate provided that it has been programmed by the federation designer to identify its own federation save criteria.

A pictorial representation of what needs to occur on each simulation execution frame in order to initiate a federation save is shown in the following figure:

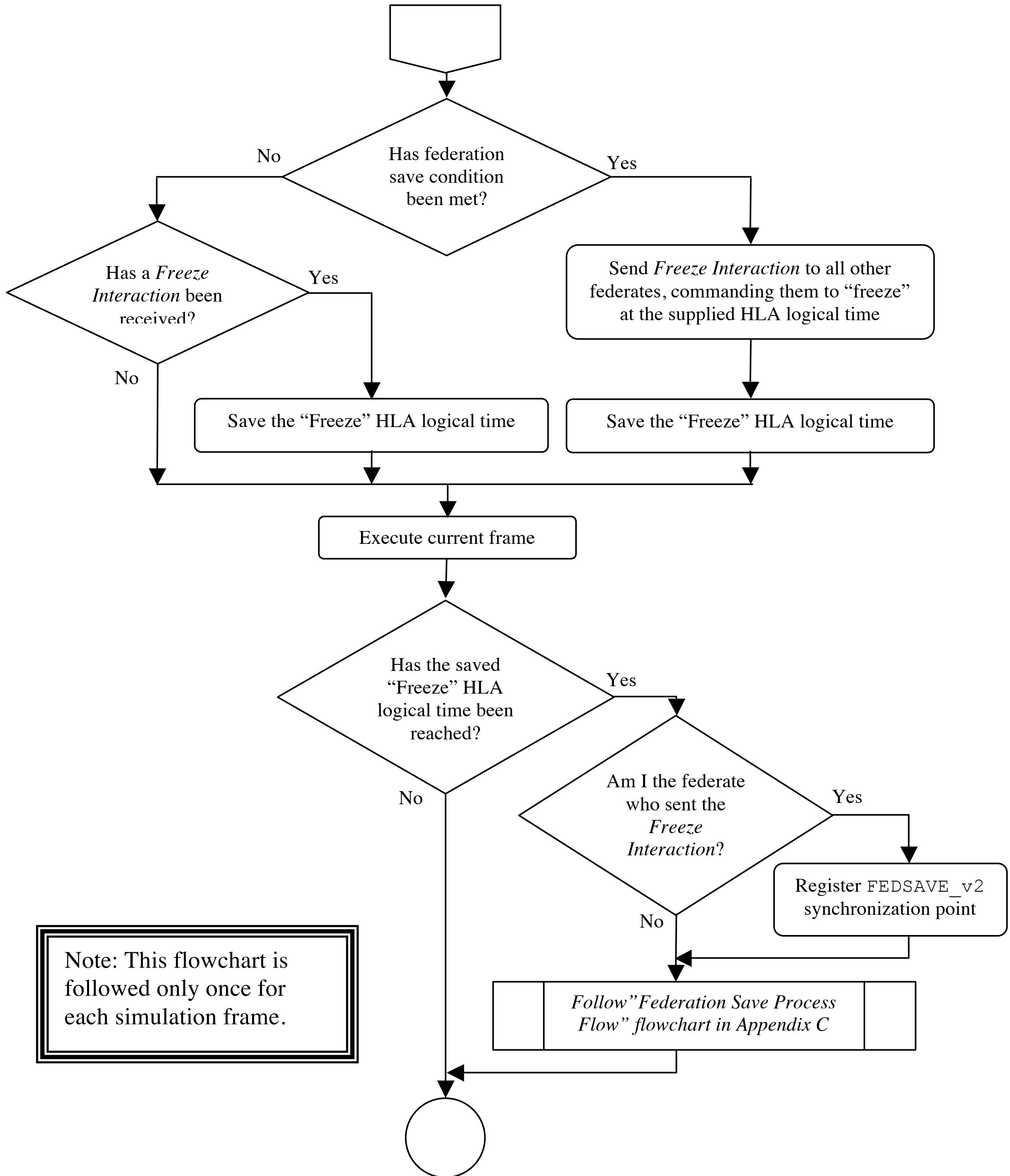


Figure 2 IMSim Federation Save High-Level Representation

To guarantee that all federates perform their save at an identical point in the federation execution, we need to coordinate all federates onto a consistent HLA logical time and frame boundaries to achieve a consistent state before initiating the federation save.

To achieve this coordination between all federates, a new TimeStamp Order (TSO) interaction shall be sent to all federates. This new TSO interaction class, called 'Freeze', must be added to the FOM file.

Otherwise, a message like the following will be emitted, “Interaction FOM Name 'Freeze' Not Found. Please check your input or modified-data files to make sure the Interaction FOM Name is correctly specified.”, when you attempt to start the federation without this new interaction defined in the FOM file.

This new interaction shall contain a single parameter, an **HLAinteger64BE** data type called ‘time’, which represents an integer time in microseconds. The ‘time’ value shall be the HLA logical time identifying the frame upon whose completion the federation will go into freeze mode.

Here is the *TrickHLAFreezeInteraction.xml* FOM Module for the *Freeze* interaction:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<objectModel xsi:schemaLocation="http://standards.ieee.org/IEEE1516-2010
   http://standards.ieee.org/downloads/1516/1516.2-2010/IEEE1516-DIF-2010.xsd"
   xmlns="http://standards.ieee.org/IEEE1516-2010"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelIdentification>
        <name></name>
        <type>FOM</type>
        <version></version>
        <securityClassification></securityClassification>
        <purpose></purpose>
        <applicationDomain></applicationDomain>
        <description></description>
        <useLimitation></useLimitation>
        <other></other>
    </modelIdentification>
    <interactions>
        <interactionClass>
            <name>HLAinteractionRoot</name>
            <interactionClass>
                <name>Freeze</name>
                <sharing>PublishSubscribe</sharing>
                <dimensions/>
                <transportation>HLAreliable</transportation>
                <order>TimeStamp</order>
                <semantics></semantics>
                <parameter>
                    <name>time</name>
                    <dataType>HLAinteger64BE</dataType>
                    <semantics></semantics>
                </parameter>
            </interactionClass>
        </interactionClass>
    </interactions>
</objectModel>
```

All federates must be upgraded to publish and unpublish, subscribe and unsubscribe this new interaction class.

When a federation save condition has been met inside a federate, this federate shall send a *Freeze Interaction* immediately to arrive on the next HLA logical frame, supplying the HLA logical time on which all federates shall suspend execution in preparation of a federation save.

An unavoidable delay is in inherent to any TSO interaction: the earliest that a federate can act upon an interaction is in the frame in which it was received. Therefore, the earliest a coordinated federation save can occur is the on the frame after the federation save condition was met.

If the user wishes to delay the federation save to occur in the future (i.e. not in the next execution frame), they need to supply a future HLA logical time into the *Freeze Interaction*.

All federates which receive the *Freeze Interaction* shall save the contained HLA logical time. The *Freeze Interaction* sending federate will not receive the interaction, therefore it must internally save the HLA logical time.

All federates shall go into 'freeze mode' only when the *Freeze Interaction*'s HLA logical time has been reached and after the frame has completed execution, not on the frame which received the *Freeze Interaction*.

Follow the procedures outlined in the next chapter.

15 Saving the federation

When a *Freeze Interaction* is received or the federate who initiated the *Freeze Interaction* has reached the *Freeze Interaction* HLA logical time, the following steps must be followed in order to successfully save a federation. This procedure is represented in the flowchart located in Appendix C.

15.1 Process / emulate the *Freeze Interaction*
15.2 Suspend / freeze execution after the completion of the frame
15.3 Acknowledge the "FEDSAVE_v2" save sync point
15.4 Wait for federation synchronization on the "FEDSAVE_v2" sync point
15.5 Request a federation save from the RTI
15.6 Wait until the federation is ready to save
15.7 Turn off all data exchange & interaction processing
15.8 Save running federate information into an external file
15.9 Convert all interactions, ownership transfers and sync points into save-able format
15.10 Signal the RTI that this federate has begun saving
15.11 Write the checkpoint file
15.12 Signal the RTI that the federate save is complete
15.13 Wait for all federates to complete their save
15.14 Decode and report all federation save failure(s)
15.15 Restart data exchange & interaction processing
15.16 Resume execution using the "FEDRUN_v2" synchronization point

15.1 Process / emulate the *Freeze Interaction*

When a *Freeze Interaction* is received, as shown in Figure 2 IMSim Federation Save High-Level Representation, necessary steps must be taken to save this HLA logical time into the federate.

If this is the federate that sent the *Freeze Interaction*, we must emulate the receipt of a *Freeze Interaction* as shown in Figure 2 IMSim Federation Save High-Level Representation, by saving the HLA logical time that was sent inside the *Freeze Interaction* into the federate.

15.2 Suspend / freeze execution after the completion of frame

As shown in Figure 2 IMSim Federation Save High-Level Representation, after each simulation frame has completed execution, a check is made if a *Freeze Interaction* HLA logical time has been reached.

This must be done so that all federates are at an identical place in the execution frame when the federation is saved.

When a *Freeze Interaction* HLA logical time has been reached, the federate shall delete it from memory and suspend execution of this federate in preparation for a federation save. If there is no matching HLA logical freeze time, jump to section 15.16 to prepare to execute the next simulation frame.

We must be aware that a race condition could occur if a federation save is initiated before all federates are in 'freeze mode'. To prevent it from occurring, the federate which sent the *Freeze Interaction* shall also register a 'FEDSAVE_v2' synchronization point with the RTI once the federate enters 'freeze mode', using the code snippet below:

```
RTIAmbassador->registerFederationSynchronizationPoint( L"FEDSAVE_v2",
                                                       UserSuppliedTag( 0, 0 ) );
```

15.3 Acknowledge the "FEDSAVE_v2" federation save sync point

Acknowledge the receipt of the synchronization point with the RTI by calling this routine:

```
RTIAmbassador->synchronizationPointAchieved( L"FEDSAVE_v2" );
```

15.4 Wait for federation to synchronize on the "FEDSAVE_v2" sync point

Wait until the federation is synchronized on the FEDSAVE_v2 label before proceeding to the request a federation save. Execute a blocking loop until the MyFedAmbassador::federationSynchronized() RTI callback is called to inform us that the federation is synchronized on this sync point.

```
void MyFedAmbassador::federationSynchronized ( wstring const & label)
throw ( RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->federation_synchronized( label );
}
```

The below code would scan for the FEDSAVE_v2 sync point label name. Once found, it will continue the federation save.

```
void MyFederate::federation_synchronized( wstring const & label )
{
   // remove the label from sync points data structure

   size_t found = label.find( L"FEDSAVE_v2" );
   if ( found != wstring::npos ) {
      // set flag to trigger the wait loop for a initiate federate save
      // RTI callback.
   }
}
```

15.5 Request a federation save from the RTI

If this federate was the one that met the condition for a federation save and sent the *Freeze Interaction*, we are the only federate which communicates with the RTI in order to initiate the federation save. All other federates would skip to the next section.

We need to ensure that a previous federation save is not queued in order to avoid issuing a duplicate federation save request. We request this information from the RTI as follows:

```
save_request_complete = false;
RTIAmbassador->queryFederationSaveStatus();
```

The RTI callback would be received in the following routine, which, in turn, instructs the federate to decode the status vector:

```
void MyFedAmbassador::federationSaveStatusResponse(
   FederateHandleSaveStatusPairVector const &theFederateStatusVector)
throw (FederateInternalError)
{
```

```
   myfederate->process_requested_federation_save_status( theFederateStatusVector );
}
```

The following routine decodes the status vector for the requesting federate since nobody else should communicate with the RTI yet. If this federate has not requested the federation save, then we will request it. Set the appropriate flag to signal the federate to initiate the federation save.

```
void MyFederate::process_requested_federation_save_status(
   auto_ptr< FederateHandleSaveStatusPairVector > theFederateStatusVector)
{
   FederateHandleSaveStatusPairVector::iterator vector_iter;

   // load the first element from 'theFederateStatusVector'.
   vector_iter = theFederateStatusVector->begin();

   // loop through all elements...
   while ( vector_iter != theFederateStatusVector->end() ) {
      // if this element's federate_id matches our federate_id, do something...
      if (federate_id.getImplementation() ==
                vector_iter->first.getImplementation()) {
         if (vector_iter->second == NO_SAVE_IN_PROGRESS) {
            initiate_save_flag = true;
         }
         break; // exit the while loop since we found our id...
      }
      // load the next element from 'theFederateStatusVector'.
      vector_iter++;
   }

   save_request_complete = true;
}
```

The federate would execute a wait loop until the status vector is received and decoded.

```
while ( ! save_request_complete ) {
   usleep ( 100 ); // sleep until RTI responds...
}
```

If the `initiate_save_flag` has been set, request the federation save from the RTI by supplying the `checkpoint_set_name` as the only parameter.

IEEE_1516.1-2000, section 4.11 describes how to request a federation save. We will utilize the single parameter version, in which "the RTI shall instruct all federation execution members to save state as soon as possible after the invocation of the Request Federation Save service"[1] Since we have chosen to perform the checkpoint in suspended execution mode, at the end of the current execution frame, the second parameter (time) must not be specified on the call as this would break the current federation save design.

```
if ( initiate_save_flag ) {
   RTIAmbassador->requestFederationSave( checkpoint_set_name );
}
```

15.6 Wait until the RTI callback informs us to save our data

All federates need to execute a wait loop until they receive the RTI callback, via the `MyFedAmbassador::initiateFederateSave()` method, which signals them to perform their save.

```
void MyFedAmbassador::initiateFederateSave(
       wstring const & label )
throw ( UnableToPerformSave,
```

```
        FederateInternalError )
{
   myfederate->set_save_name( label );
   myfederate->set_start_to_save( true );
   myfederate->set_save_begun();
}
```

Add a blocking loop to `MyFederate::perform_checkpoint()` method to wait until the `start_to_save` flag has been set.

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   :
   : the rest of the checkpoint point code
   :
}
```

15.7 Turn off all data exchange and interaction processing

Per IEEE_1516.1-2000, section 4.12, "the joined federate shall stop providing new information to the federation immediately after receiving the Initiate Federate Save service invocation. The joined federate may resume providing new information to the federation only after receiving the Federation Saved service invocation."[1]

The simulation designer is responsible for turning off the sending and receiving of data and the processing of interactions in all federates before the federate save is initiated.

15.8 Save running federate information into an external file

We need to save a snapshot of which federates are joined and running at the time of the checkpoint so we know what federates to allow to rejoin when we request a federation restore in a subsequent federation run.

This information should be saved by all federates into a file external to the checkpoint file, saved in the same directory as the checkpoint file. This is done to speedup the restore process by loading this small file verses loading the potentially huge checkpoint file just to gain access to the list of federates that were running at the time the checkpoint file was created.

Name the file the `checkpoint_set_name` with '`.running_feds`' appended to it (i.e., if the `checkpoint_set_name` is `foo` then this file name would be `foo.running_feds`).

This information was retrieved before the federate turned on time management (see section 8.8), and is kept current throughout the lifetime of the federate with the information from all federates who join or resign from the federation (via RTI callbacks via `MyFedAmb::discoverObjectInstance()` and `MyFedAmb::removeObjectInstance()`, respectfully). Appropriate action should be taken by all federates to keep this list current when receiving the above RTI callbacks.

For each federate that is currently joined we need to save: 1) the federate name (the name given to the federation upon joining – retrieved by the `MyFederate::set_federate_instance_name()` routine in section 6.1), 2) the federate's MOM name (supplied to us by the RTI as the 3rd parameter [object instance name] on the `MyFedAmb::discoverObjectInstance()` callback), and 3) a flag to indicate that this federate is required, set to true.

The federate name will be used to guarantee that only these required federates rejoin the federation when it is restored from the checkpoint set. The federate's MOM name is needed to re-establish the handle with the RTI since this is the name that it was given by the RTI at federation startup and the RTI, henceforth, would recognize this federate only by this handle name.

The creation of the '`.running_feds`' file is shown in the below routine.

```
void MyFederate:: write_running_feds_file( const char * file_name )
{
   char full_path[ 1024 ];
   ofstream file;

   sprintf( full_path, "%s/%s.running_feds",this->save_directory, file_name );
   file.open(full_path, ios::out);
   if (file.is_open()) {
      file << running_feds_count << endl;

      // echo the contents of running_feds into file...
      for ( int i = 0; i < running_feds_count; i++ ) {
         file << running_feds[i].MOM_instance_name << endl;
         file << running_feds[i].name << endl;
         file << running_feds[i].required << endl;
      }

      file.close(); // close the file.

   } else {
      fatalError( "MyFederate::write_running_feds_file() => failed to "
                  "open file '%s' for writing!", full_path );
   }
}
```

Call the above routine from `perform_checkpoint()` routine, as shown below:

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   // section 15.7: turn of all data exchange and interaction processing here

   // section 15.8: write the '.running_feds' file
   write_running_feds_file( checkpoint_set_name );

   :
   : the rest of the checkpoint point code
   :
}
```

15.9 Convert all interactions, ownership transfers and sync points into save-able format

All federate data must be converted to primitive data types that can be saved to disk.

The IEEE_1516.1-2000, section 1.4.3 defines a RTI handle as "capable of being manipulated by a computer or for communication between a federate and the RTI. Originated by the RTI, federation execution-wide unique, and unpredictable." [1]. Thus, they are immutable, and are not checkpoint-able because their value is "unpredictable". We need to save into the checkpoint file enough unique

information for each RTI handle with which we can request each handle back from the RTI upon the restore of this federate.

We need to convert all pending interactions, pending ownership transfer commands and all future synchronization points that the federate contains into save-able values. This must be done so that we can restore this data back into memory and re-enable any pending commands upon the federation restore.

The placement of the above is added to the `perform_checkpoint()` routine, as shown below:

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   // section 15.7: turn of all data exchange and interaction processing here

   // section 15.8: write the '.running_feds' file
   write_running_feds_file( checkpoint_set_name );

   // section 15.9: convert all interactions, ownership transfers and sync
   // points into checkpointable format here

   :
   : the rest of the checkpoint point code
   :
}
```

15.10 Signal the RTI that this federate has begun saving

Now that we are ready to save the federate, signaling to the RTI that the federate has begun saving is added to the `perform_checkpoint()` routine, as shown below:

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   // section 15.7: turn of all data exchange and interaction processing here

   // section 15.8: write the '.running_feds' file
   write_running_feds_file( checkpoint_set_name );

   // section 15.9: convert all interactions, ownership transfers and sync
   // points into checkpointable format here

   // section 15.10: tell the RTI that I am beginning my save
   try {
      this->federate_saved = false;
      RTI_amb->federateSaveBegun();

   } catch ( SaveNotInitiated ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: SaveNotInitiated"
           << endl;
    } catch ( FederateNotExecutionMember ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: "
           << "FederateNotExecutionMember " << endl;
   } catch ( RestoreInProgress ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: RestoreInProgress "
```

```
           << endl;
   } catch ( RTIinternalError &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTIinternalError: "
            << e.what()<< endl;
   } catch (RTI::exception &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTI::exception: "
            << e.what() << endl;
   }

   :
   : the rest of the checkpoint point code
   :
}
```

15.11 Write the checkpoint file

It is up to the simulation designer to write code to save all of the simulation data into a checkpoint file and trigger it from the `perform_checkpoint()` routine, as shown below:

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   // section 15.7: turn of all data exchange and interaction processing here

   // section 15.8: write the '.running_feds' file
   write_running_feds_file( checkpoint_set_name );

   // section 15.9: convert all interactions, ownership transfers and sync
   // points into checkpointable format here

   // section 15.10: tell the RTI that I am beginning my save
   try {
      this->federate_saved = false;
      RTI_amb->federateSaveBegun();

   } catch ( SaveNotInitiated ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: SaveNotInitiated"
           << endl;
    } catch ( FederateNotExecutionMember ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: "
           << "FederateNotExecutionMember " << endl;
   } catch ( RestoreInProgress ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: RestoreInProgress "
           << endl;
   } catch ( RTIinternalError &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTIinternalError: "
            << e.what() << endl;
   } catch (RTI::exception &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTI::exception: "
            << e.what() << endl;
   }

   // section 15.11: write the checkpoint file here

   :
   : the rest of the checkpoint point code
   :
}
```

15.12 Signal the RTI that the federate save has completed saving

Once the federate has finished creating the checkpoint file, add code to the `perform_checkpoint()` method to inform the RTI of the federate's success or failure writing the checkpoint file, as shown below:

```
void MyFederate::perform_checkpoint()
{
   // section 15.6: wait until we receive the signal to start saving
   if ( ! this->start_to_save ) {
      usleep ( 100 ); // sleep until RTI responds...
   }

   // section 15.7: turn of all data exchange and interaction processing here

   // section 15.8: write the '.running_feds' file
   write_running_feds_file( checkpoint_set_name );

   // section 15.9: convert all interactions, ownership transfers and sync
   // points into checkpointable format here

   // section 15.10: tell the RTI that I am beginning my save
   try {
      this->federate_saved = false;
      RTI_amb->federateSaveBegun();

   } catch ( SaveNotInitiated ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: SaveNotInitiated"
           << endl;
    } catch ( FederateNotExecutionMember ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: "
           << "FederateNotExecutionMember " << endl;
   } catch ( RestoreInProgress ) {
      cout << "MyFederate::perform_checkpoint() EXCEPTION 1: RestoreInProgress "
           << endl;
   } catch ( RTIinternalError &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTIinternalError: "
            << e.what() << endl;
   } catch (RTI::exception &e ) {
      wcout << L"MyFederate::perform_checkpoint() EXCEPTION 1: RTI::exception: "
            << e.what() << endl;
   }

   // section 15.11: write the checkpoint file here

   // section 15.12: inform the RTI of success / failure of checkpoint file write
   if ( status_successful ) {
      try {
         RTI_amb->federateSaveComplete();

         this->federate_saved = true;
         this->start_to_save = false;
         this->announce_save = false; // turn off the flag so we do not try
                                      // to perform a checkpoint again...
      } catch ( FederateHasNotBegunSave ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "FederateHasNotBegunSave " << endl;
      } catch ( FederateNotExecutionMember ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "FederateNotExecutionMember " << endl;
      } catch ( RestoreInProgress ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "RestoreInProgress " << endl;
      } catch ( RTIinternalError &e ) {
         wcout << L"MyFederate::perform_checkpoint() EXCEPTION 2: "
```

```
               << L"RTIinternalError: " << e.what() << endl;
      } catch ( RTI::exception &e ) {
         wcout << L"MyFederate::perform_checkpoint() EXCEPTION 2: "
               << L"RTI::exception: " << e.what() << endl;
      }
   } else {
      try {
         RTI_amb->federateSaveNotComplete();

         this->federate_saved = true;
         this->start_to_save = false;
         this->announce_save = false; // turn off the flag so we do not try
                                      // to perform a checkpoint again...
      } catch ( FederateHasNotBegunSave ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "FederateHasNotBegunSave " << endl;
      } catch ( FederateNotExecutionMember ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "FederateNotExecutionMember " << endl;
      } catch ( RestoreInProgress ) {
         cout << "MyFederate::perform_checkpoint() EXCEPTION 2: "
              << "RestoreInProgress " << endl;
      } catch ( RTIinternalError &e ) {
         wcout << L"MyFederate::perform_checkpoint() EXCEPTION 2: "
               << L"RTIinternalError: " << e.what() << endl;
      } catch ( RTI::exception &e ) {
         wcout << L"MyFederate::perform_checkpoint() EXCEPTION 2: "
               << L"RTI::exception: " << e.what() << endl;
      }
   }

// end of the perform_checkpoint routine
}
```

15.13 Wait for all federates to complete their save

Execute a wait loop until the RTI will informs the us that the federation save has successfully completed or failed, via the RTI callback routines `MyFedAmbassador::federationSaved()` and `MyFedAmbassador::federationNotSaved()`, respectfully.

If the federation was successfully saved, we need to reset the state of the federate so it can return to an executing state.

```
void MyFedAmbassador::federationSaved()
throw ( RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->set_start_to_save( false );
   myfederate->set_save_completed( true );
   myfederate->federation_saved();
}
```

Turn off all federate flags associated with the saving the federate.

```
void MyFederate::federation_saved()
{
    // reset all values / status flags associated with a federate save
}
```

15.14 Decode and report reason(s) for federation save failure

If the federation save was not successful, we would receive a RTI callback via the `MyFedAmbassador::federationNotSaved()` method, providing a `SaveFailureReason` object containing the reason(s) why the federation save failed. Have this method call a routine inside

your Federate to decode the reason(s) of the failure, like the following code snippet, and reset the state of the federate so it can return to an executing state.

```
void MyFedAmbassador::federationNotSaved(SaveFailureReason saveFailureReason)
throw ( RTI1516_NAMESPACE::FederateInternalError )
{
   myfederate->decode_save_failure_reason( saveFailureReason );

   myfederate->set_start_to_save( false );
   myfederate->set_save_completed( true );
   myfederate->federation_saved();
}
```

```
void MyFederate::decode_save_failure_reason( SaveFailureReason theReason )
{
   if (theReason == RTI_UNABLE_TO_SAVE ) {
      cout << "The federation failed to save. reason=\""
           << "RTI_UNABLE_TO_SAVE\"" << endl;
   }
   if (theReason == FEDERATE_REPORTED_FAILURE_DURING_SAVE ) {
      cout << "The federation failed to save. reason=\""
           << "FEDERATE_REPORTED_FAILURE_DURING_SAVE\"" << endl;
   }
   if (theReason == FEDERATE_RESIGNED_DURING_SAVE ) {
      cout << "The federation failed to save. reason=\""
           << "FEDERATE_RESIGNED_DURING_SAVE\"" << endl;
   }
   if (theReason == RTI_DETECTED_FAILURE_DURING_SAVE ) {
      cout << "The federation failed to save. reason=\""
           << "RTI_DETECTED_FAILURE_DURING_SAVE\"" << endl;
   }
   if (theReason == SAVE_TIME_CANNOT_BE_HONORED ) {
      cout << "The federation failed to save. reason=\""
           << "SAVE_TIME_CANNOT_BE_HONORED\"" << endl;
   }
}
```

15.15 Restart data exchange & interaction processing

Since we have completed the federation save, we need to re-enable the exchange of data and processing of interactions, which were turned off in section 15.7.

15.16 Resume execution using the "FEDRUN_v2" synchronization point

Now that we have completed the task of a federation save, we shall resume execution of the simulation by using the `FEDRUN_v2` synchronization point to coordinate all the federates going back to run.

15.16.1 Registering the "FEDRUN_v2" synchronization point

Any of the joined federates can initiate the return to run by registering the `FEDRUN_v2` synchronization point, but typically the federate that initiated the save will initiate the return to run. A federate will call the RTI API to register the `FEDRUN_v2` synchronization point, which will result in it being announced to all the other federates.

15.16.2 Achieve the "FEDRUN_v2" synchronization point

When the `FEDRUN_v2` synchronization point is announced through a Federate Ambassador callback, the federate will call the RTI API to achieve the synchronization point.

15.16.3 Wait for the federation to be synchronized on "FEDRUN_v2"

All federates will wait for the Federate Ambassador callback indicating the federation is synchronized on the `FEDRUN_v2` synchronization point. Once the federation is synchronized, all federates will to go run by existing 'freeze mode'.

16 How to resign from the federation with the intention of rejoining

If the federation designer has decided that a federate should have the capability of rejoining the federation within the lifetime of the federation, a few items must be addressed.

First, the federate must resign in such a manner that makes rejoining the federation possible. The resigning federate must divest ownership of it attributes when resigning without deleting itself from the federation. This is accomplished by resigning with option 1, "Unconditionally divest ownership of all owned instance attributes", discussed in Section 4.5 of the IEEE 1516 standards [1]. If the user invokes resign options 2, 4 or 5, the RTI will implicitly invoke the `Delete Object Instance` callback in the remaining federates forcing the deletion of the instance attributes from the federation [1], thus eliminating the possibility from ever rejoining the federation.

Second, the federation designer must decide which of the remaining federate(s) would take ownership of the attributes once the federate resigns. If this is not done, the object containing the attributes will become a federation orphan and cannot be re-discovered by the remaining federates by any means, as discussed in Section 6.1, paragraph 11, of the IEEE 1516 standards [1]. To prevent this object from becoming a federation orphan, update the remaining federate(s) to publish at least one of the resigning federate's attributes. The federate(s) that publish the aforementioned attributes will receive RTI callbacks to acquire of ownership of the attributes, which it publishes. Additionally, the federate(s) which publish the aforementioned attributes – and actually do gain ownership of the attributes while the resigned federate does exist – must divest ownership of the attributes back to the rejoined federate when it requests ownership when it rejoins the federation.

Note: It is beyond the scope of this document to show the user how to divest / acquire ownership of attributes. This is covered in Chapter 7 of the IEEE 1516 standards [1].

17 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.

[3] Daniel E. Dexter. *Distributed Space Exploration Simulation Multiphase Initialization Design*. National Aeronautics and Space Administration; Johnson Space Center; Software, Robotics & Simulation Division; Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, December 2007.

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].

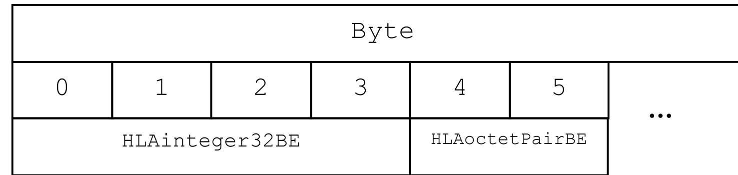


Figure 3 HLAunicodeString Format

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

Byte	0	1	2	3	4	5	6	7	8	9
Decimal Value	0	0	0	3	0	67	0	69	0	86
	length = 3					C		E		V

Figure 4 "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 HLAhandle

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

The `HLAhandle` is defined in IEEE Standard 1516.1-2000, Section 11.6 (MOM OMT tables), Table 12 as being encoded as `HLAvariableArray` and contains elements that are a list of encoded handles [1].

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 `HLAbyte` followed by the encoding of each element in sequence [2].

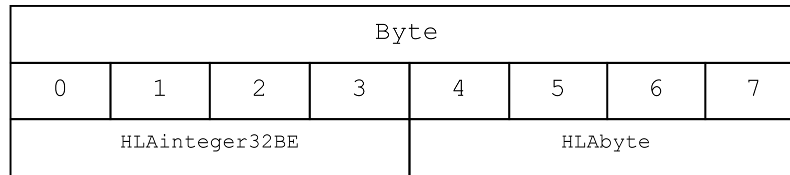


Figure 5 HLAhandle Format

To make things a little clearer, here is an example of a federate id 2 in the `HLAbyte` format.

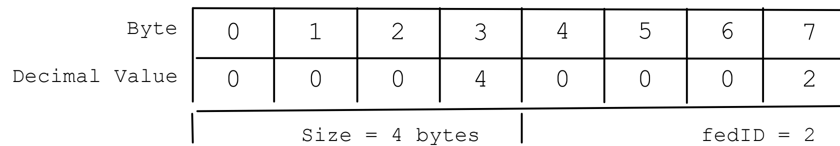
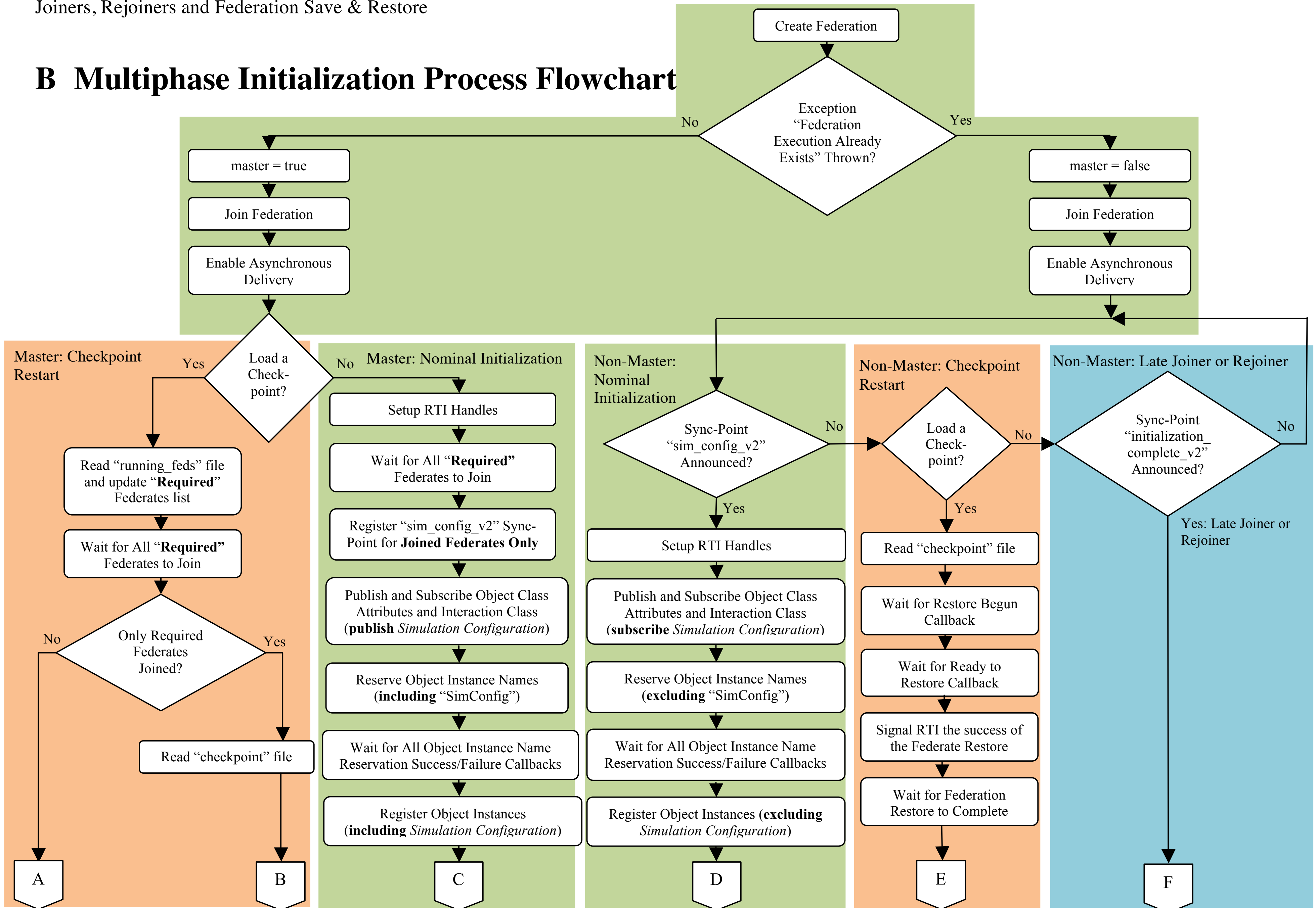


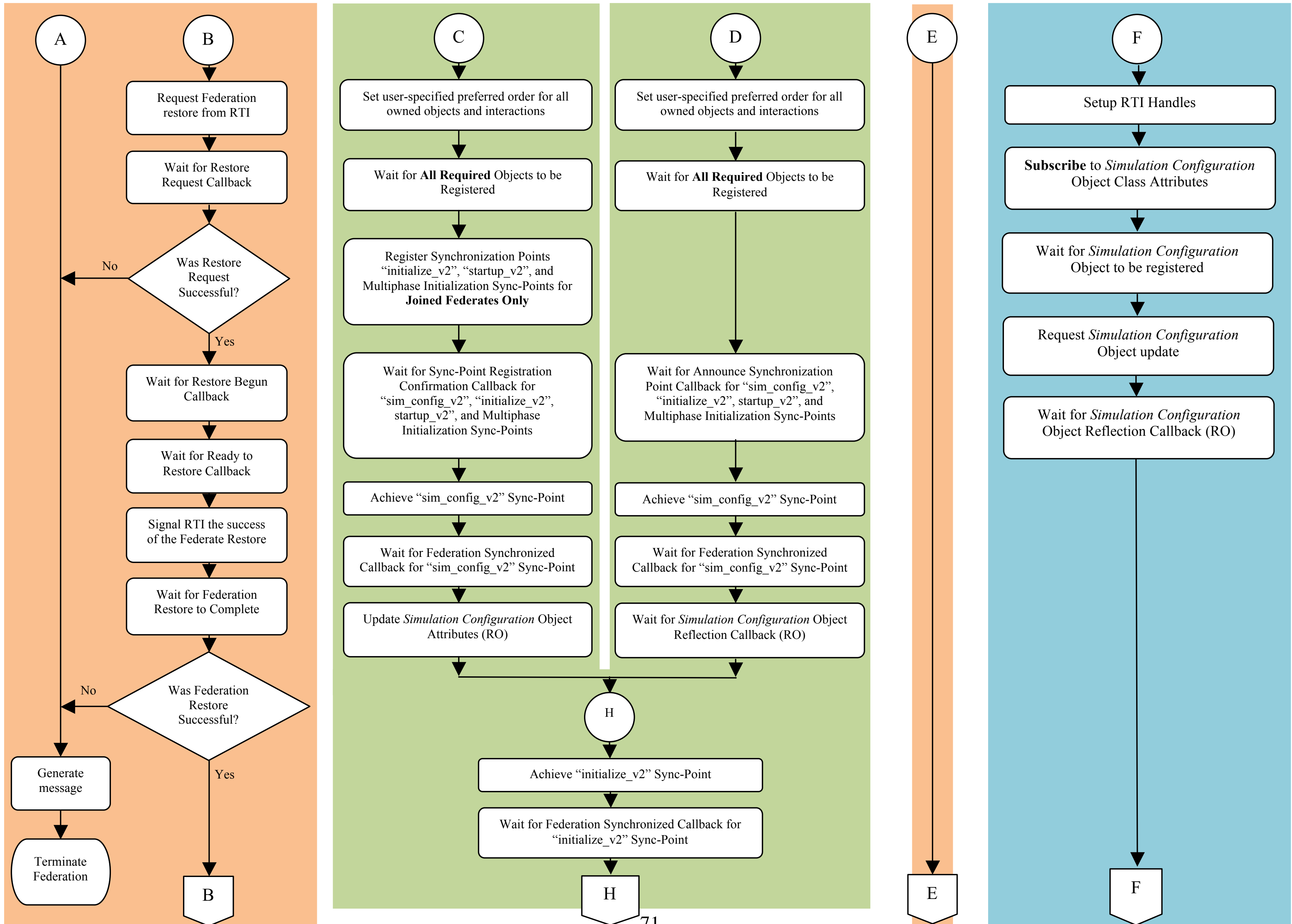
Figure 6 Example of the HLAbyte Format

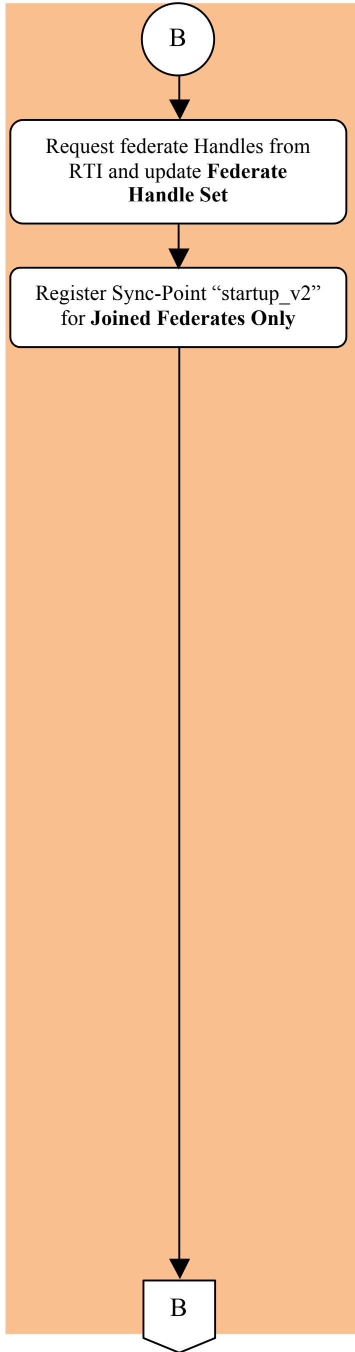
A.5 Time Objects

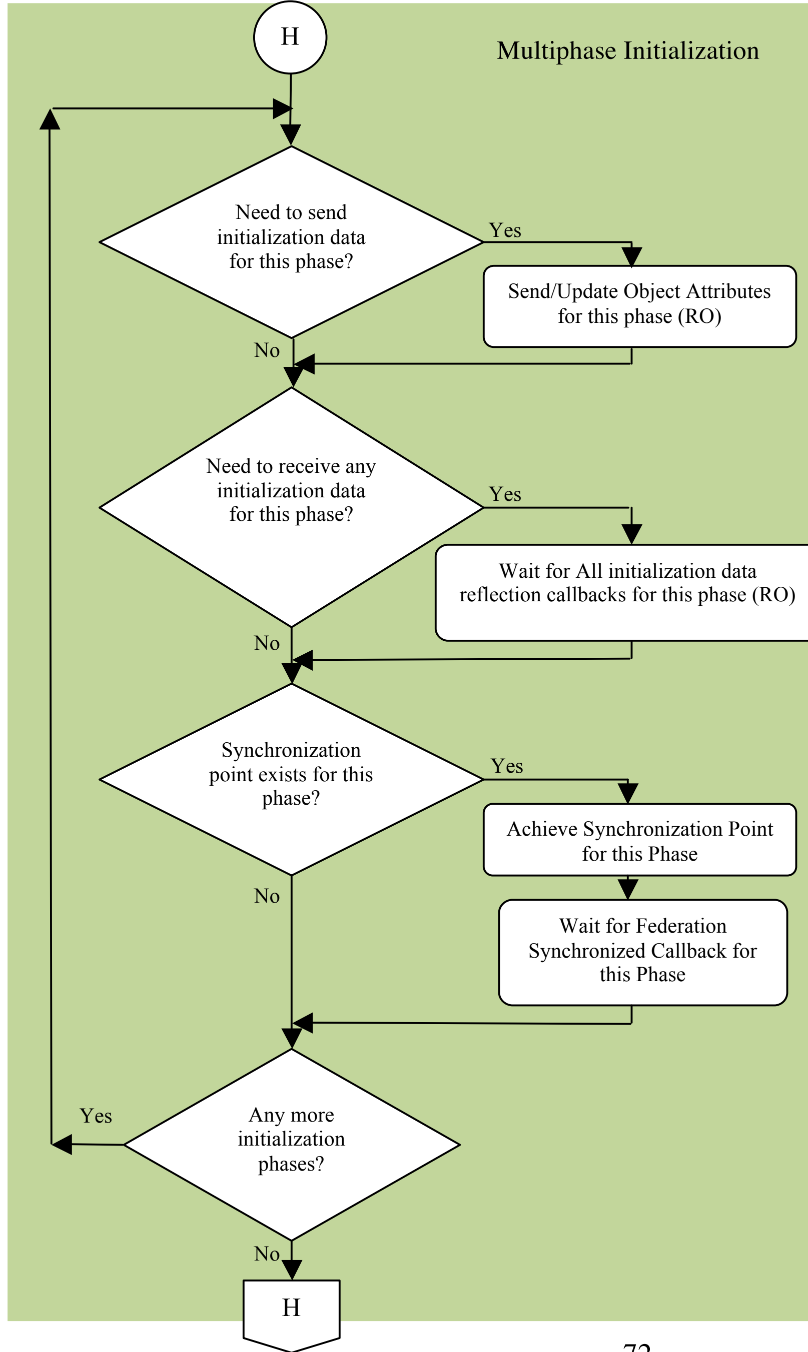
The HLA uses two abstract object classes, **LogicalTime**, and **LogicalTimeInterval**, to manage time. The Pitch RTI includes two subclasses that can be used for time encoded as doubles: **LogicalTimeDouble** and **LogicalTimeIntervalDouble**. The IMSim project has created their own class versions, 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.

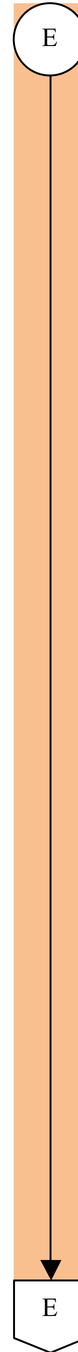
B Multiphase Initialization Process Flowchart

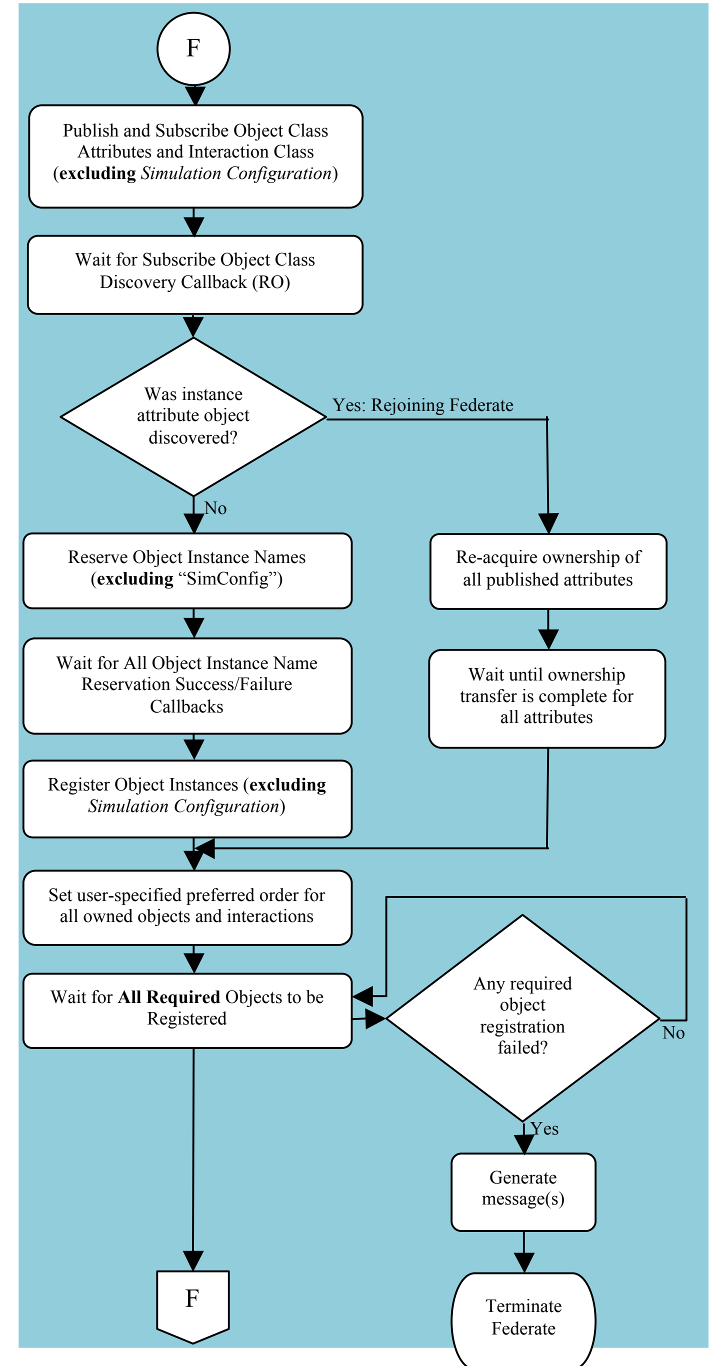


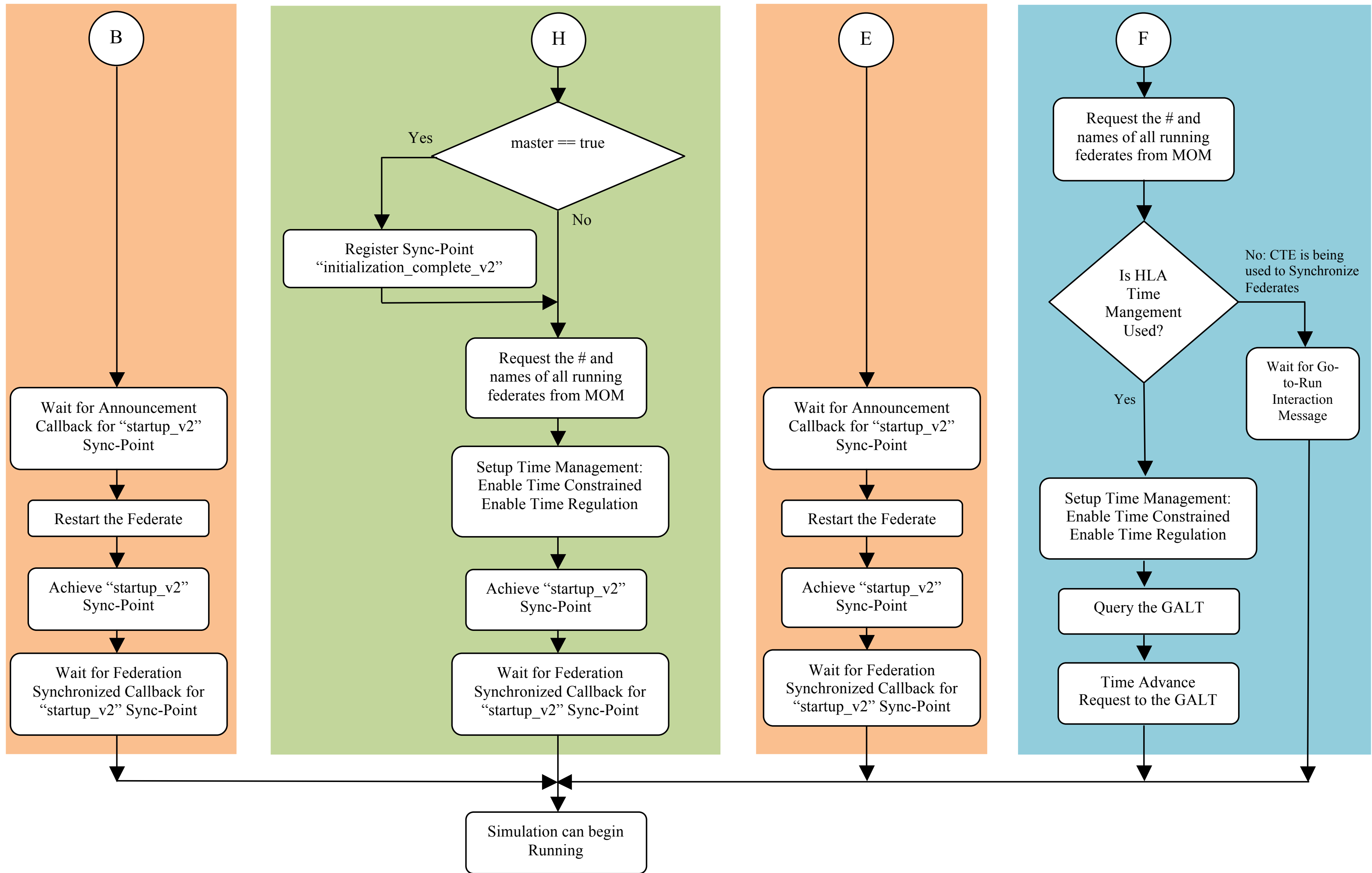

A
B
Request Federation restore from RTI
Wait for Restore Request Callback
Was Restore Request Successful?
No
Yes
Wait for Restore Begun Callback
Wait for Ready to Restore Callback
Signal RTI the success of the Federate Restore
Wait for Federation Restore to Complete
Was Federation Restore Successful?
No
Yes
Generate message
Terminate Federation
B
C
Set user-specified preferred order for all owned objects and interactions
Wait for All Required Objects to be Registered
Register Synchronization Points "initialize_v2", "startup_v2", and Multiphase Initialization Sync-Points for Joined Federates Only
Wait for Sync-Point Registration Confirmation Callback for "sim_config_v2", "initialize_v2", startup_v2", and Multiphase Initialization Sync-Points
Achieve "sim_config_v2" Sync-Point
Wait for Federation Synchronized Callback for "sim_config_v2" Sync-Point
Update Simulation Configuration Object Attributes (RO)
D
Set user-specified preferred order for all owned objects and interactions
Wait for All Required Objects to be Registered
Wait for Announce Synchronization Point Callback for "sim_config_v2", "initialize_v2", startup_v2", and Multiphase Initialization Sync-Points
Achieve "sim_config_v2" Sync-Point
Wait for Federation Synchronized Callback for "sim_config_v2" Sync-Point
Wait for Simulation Configuration Object Reflection Callback (RO)
H
Achieve "initialize_v2" Sync-Point
Wait for Federation Synchronized Callback for "initialize_v2" Sync-Point
H
E
E
F
Setup RTI Handles
Subscribe to Simulation Configuration Object Class Attributes
Wait for Simulation Configuration Object to be registered
Request Simulation Configuration Object update
Wait for Simulation Configuration Object Reflection Callback (RO)
F


B
Request federate Handles from RTI and update **Federate Handle Set**
Register Sync-Point “startup_v2” for **Joined Federates Only**
B


H
Multiphase Initialization
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
Any more initialization phases?
Yes
No
H


E
E


F
Publish and Subscribe Object Class Attributes and Interaction Class (**excluding** *Simulation Configuration*)
Wait for Subscribe Object Class Discovery Callback (RO)
Was instance attribute object discovered?
Yes: Rejoining Federate
No
Reserve Object Instance Names (**excluding** “SimConfig”)
Wait for All Object Instance Name Reservation Success/Failure Callbacks
Register Object Instances (**excluding** *Simulation Configuration*)
Re-acquire ownership of all published attributes
Wait until ownership transfer is complete for all attributes
Set user-specified preferred order for all owned objects and interactions
Wait for **All Required** Objects to be Registered
Any required object registration failed?
No
Yes
Generate message(s)
Terminate Federate
F


B
H
E
F
master == true
Yes
No
Register Sync-Point
“initialization_complete_v2”
Request the # and
names of all running
federates from MOM
Setup Time Management:
Enable Time Constrained
Enable Time Regulation
Achieve “startup_v2”
Sync-Point
Wait for Federation
Synchronized Callback for
“startup_v2” Sync-Point
Wait for Announcement
Callback for “startup_v2”
Sync-Point
Restart the Federate
Achieve “startup_v2”
Sync-Point
Wait for Federation
Synchronized Callback for
“startup_v2” Sync-Point
Wait for Announcement
Callback for “startup_v2”
Sync-Point
Restart the Federate
Achieve “startup_v2”
Sync-Point
Wait for Federation
Synchronized Callback for
“startup_v2” Sync-Point
Request the # and
names of all running
federates from MOM
Is HLA
Time
Mangement
Used?
No: CTE is being
used to Synchronize
Federates
Wait for Go-
to-Run
Interaction
Message
Yes
Setup Time Management:
Enable Time Constrained
Enable Time Regulation
Query the GALT
Time Advance
Request to the GALT
Simulation can begin
Running

C Federation Save Process Flowchart