5.0 Declaration Management

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support declaration management. Declaration management includes publication, subscription and supporting control functions. Federates that produce objects (or object attributes) or that produce interactions must declare exactly what they are able to publish (i.e., generate). Federates that consume objects (or object attributes) or that consume interactions must declare their subscription interests. (See Figure 5.1).

---

**Declaration Management**

- Declaration Management
  - Coordinates data exchange between federates.
  - Specifies the data a federate will send and receive.
  - Controls where data is sent based on external interest.

---

*Figure 5.1 Declaration Management*
Declaration Management

• Declaration Management includes publication, subscription, and supporting control functions.
  - Federates that produce objects (or object parts) or that produce interactions must declare exactly what they are able to publish (i.e. generate and provide to the federation.)
  - To begin receiving updates of an attribute or an interaction, a federate must declare (subscribe) its interest in the attribute.

Figure 5.1 Declaration Management (2)

The RTI uses this information to match what is being published to information which has been subscribed to by other federates. This matching enables the RTI to tell producing federates whether to bother updating an attribute or producing interactions of a given class.

Figure 5.2 Declaration Management
The RTI keeps track of what participating federates can produce and what they are interested in consuming and sends control signals to throttle what is produced based on consumer interest. As depicted in Figure 5.2, the RTI uses control signals to inform producers exactly what they should transmit. The goal is to keep traffic off the communications network and off nodes supporting federates that don’t need the data.

5.1 Object Vocabulary Review

It is worth to take a moment for a brief review of some basic HLA terminology. HLA objects refer to simulated entities that persist (or endure) for some interval of simulated time. The OMT defines classes and objects. The HLA specification defines an object class formally as a fundamental element of a conceptual representation for a federate that reflects the real world at levels of abstraction and resolution appropriate for federate interoperability. Each object class has a name and defines a (possibly empty!) set of named data called attributes that are supposed to characterize that class in a certain way. A unique instantiation of an object class that is independent of all other instances of that class is called an object instance. Thus an attribute can be defined as a named characteristic of an object class or an object instance. Informally (and somewhat imprecisely) speaking, one can say that object classes are comprised of attributes.

For example, traffic light might be an object class. Objects of type traffic light have certain attributes (e.g., color, size, and duration). Actual, simulated traffic lights are instances of the object class traffic light. The term "object" standing alone is sometimes used to describe an instance of a particular object class, but sometimes refers to the type information. Object classes may be related to cookie cutters and object instances to the cookies produced using the cookie cutters.
Object Vocabulary

- **Object Classes**
  - Comprised of attributes
  - Describe types of things that can **persist**

- **Interaction Classes**
  - Comprised of parameters
  - Describe types of events (transitory)

Objects persist, interactions do not

---

**Figure 5.3 Object Vocabulary Review**

An *interaction* is an explicit action taken by a federate that may have some effect or impact on another federate within a federation execution. The information associated with an interaction is contained in a set of named data called *parameters*. A federate *sends* an interaction (through the RTI) to other federates that *receive* it. The interaction parameters contain the information that a federate potentially affected by the interaction may receive in order to calculate the effects of that interaction on its current state. The interaction has no continued existence after it has been received. Thus one can say that an interaction *occurs at a point* in a simulated time; unlike an object, it *does not persist* over an interval of simulated time. The OMT defines *interaction classes* as templates for a set of characteristics that are common to a group of interactions. Informally speaking again, one can say that interaction classes are *comprised of* parameters. It's fair to say, *objects are similar to interactions in so much as objects are comprised of attributes, and interactions are comprised of parameters*. The HLA recognizes this inherent symmetry and leverages it when appropriate. The primary difference between objects and interactions is *persistence*. Objects persist, interactions do not (See Figure 5.3 for a summary.)

Would an airliner be described by an object class or an interaction class? The answer depends on the purpose of the simulation. A simulation that focuses on tourism may have no interest in the flight characteristics of the airliner itself, but only in the number of tourists that arrive on a given flight. The dynamics of the flight may be of no importance whatever to the simulation. Here, the flight could be modeled as an interaction – possibly between the countries of departure and arrival. Another
simulation may focus on the in-flight characteristics of the airliner. The fact that the airliner carries tourists may be incidental. Here, the airliner persists and should be modeled as an object.

5.2 Object Hierarchies

Figure 5.4 illustrates a class hierarchy and accompanying Venn diagram. Object classes and interaction classes can be constructed hierarchically. For example, assume that objects of type $W$ are comprised of the attributes "a," "b," "c," and "d" – abbreviated \{a, b, c, d\}. It is possible to define object classes that extend object class $W$. Object class $W$ is extended to produce the object classes $X$ and $Z$. Object class $X$ is further extended to produce the object class $Y$.

![Class Hierarchy -- Venn Diagram](image)

Figure 5.4 Class Hierarchy – Venn Diagram
Object-oriented programming enthusiasts will recognize such hierarchical representations. Various communities use different phrases to describe object hierarchies. Some examples include:

- \(X\) extends \(W\).
- \(W\) is a base type.
- \(X\) is derived from \(W\).
- \(Y\) is a descendant of \(W\).
- \(W\) is the parent of \(Z\).
- \(Y\) inherits from \(X\).
- \(W\) is an ancestor of \(Z\).
- \(X\) is a child of \(W\).
- \(Y\) and \(Z\) are leaf objects.

The basic idea is that when an object class is extended to produce a new object class, the new object class contains all the attributes of the class being extended and possibly more. The object diagram and Venn diagram (Figure 5.4) illustrate the relationship between the object classes \(W, X, Y,\) and \(Z\). Object class \(W\) has the four attributes \(\{a, b, c, d\}\), class \(X\) adds the attributes \(\{e, f, g\}\) so instances of class \(X\) have attributes \(\{a, b, c, d, e, f, g\}\).

5.3 Publishing and Subscribing

Each federate must publish the object classes and interaction classes it plans to produce. It is possible for a federate to publish a subset of the available attributes for a given object class.

5.3.1 Object Publication

The object class \(Y\) contains the attributes \(\{a, b, c, d, e, f, g, h\}\). A federate can create instances of object class \(Y\), without specifying all of the attributes associated with \(Y\). For example, \(Y\) might be a particular kind of aircraft. A given federate may know some information about aircraft instances (e.g., position information), but rely on other federates to "fill in" the missing pieces (e.g., intelligence about the aircraft). In such a case, the federate would indicate that it can publish particular attributes associated with \(Y\). In Figure 5.5, Federate #1 indicates that it can publish attributes \(\{b, e, f, g, h\}\) for object class \(Y\).

Each federate must indicate explicitly which attributes it can produce (i.e., introduce or update) on a per class basis. Multiple federates may be able to publish \(Y\) instances. A federate might publish all of the attributes associated with object class \(Y\). Another federate may be able to publish attributes \(\{a, c, f\}\) for \(Y\).

---

1 Developers with a strong object-orientation should note that HLA "objects" are defined primarily by their constituent data elements rather than on behavior. In this way, HLA "objects" have more in common with relational models than object-oriented models.
All object classes have an available attribute called \textit{HLAprivilegeToDeleteObject}. As with all other available attributes, a joined federate will publish the \textit{HLAprivilegeToDeleteObject} class attribute at the known class of an object instance in order to be eligible for ownership of a corresponding \textit{HLAprivilegeToDeleteObject} instance attribute.

If the \textit{HLAprivilegeToDeleteObject} class attribute is not published explicitly (nor has been explicitly published since publication of the object class was most recently established), this attribute is considered to be published \textit{implicitly} if and only if it has not been explicitly unpublished by the joined federate (since publication of the object class was most recently established) and at least one attribute of the object class is explicitly published by the joined federate. A \textit{HLAprivilegeToDeleteObject} instance attribute is transferable among joined federates. While ownership of a typical instance attribute gives a joined federate the privilege to provide values for that instance attribute, ownership of the \textit{HLAprivilegeToDeleteObject} instance attribute of an object instance gives the joined federate the additional right to delete that object instance from the federation execution.

Only the federate that created a particular object instance is allowed to delete the instance unless the privilege to delete is conveyed to another federate. \textit{Ownership Management}, services provide the ability to exchange attribute update and object deletion responsibility among federates.

A federate must explicitly state every object class it intends to produce via the \textit{RTI_RTIambassador’s} method \textit{publishObjectClassAttributes()}. A separate call to \textit{publishObjectClassAttributes()} is required for every object class including objects that appear in class hierarchies. If a federate wishes to produce instances of object classes \textit{W}, \textit{X}, \textit{Y}, and \textit{Z} (Figure 5.4), it must say so explicitly using four publication calls.
5.3.2 Interaction Publication

As with object classes, each federate must state explicitly which interaction classes it intends to produce using the `publishInteractionClass()` method (Figure 5.6). Interactions are produced as "all or nothing." It isn't possible to specify which parameters in an interaction will be published. If a federate indicates that it intends to publish an interaction, it should be capable of specifying all parameters associated with the interaction.
Interaction Publication

- As with object classes, each federate must state explicitly which interaction classes it intends to produce.
- Interactions are produced as "all or nothing"
  - It is not possible to specify which parameters in an interaction will be published.

5.3.3 Object Subscription

Federates indicate their interest in certain object classes via the RTI_RTIambassador’s method `subscribeObjectClassAttributes()`. Object subscription differs from object publication. When a federate subscribes to an object class, it is expressing an interest in learning about all object instances of the class. For example, a federate subscribing to object class X (as shown in Figure 5.4) will discover all instances of class X produced by other federates in the federation. Additionally, a federate subscribing to X will discover all instances of class Y (produced by other federates) as though they were instances of class X. This is an example of type promotion, shown in Figure 5.7.

Whenever a federate expresses a subscription interest in a particular object class, the RTI presumes that the federate is interested in instances of the descendant classes as well. A federate subscribing to class W would see external instances of classes X, Y, and Z as instances of class W. This can be a useful tool. Class W might represent all aircraft. Class X might represent commercial aircraft, while class Y represents small private aircraft. A federate may wish to know about all aircraft, but not care about the details – including the private vs. commercial designation.
A federate is informed about a new object instance if (a) the federate has subscribed to the object class of the instance or (b) the instance can be promoted (i.e., up the hierarchy) to a subscribed object class. When an object is promoted, attributes particular to the original class are dropped. An instance of object class Y has attributes \{a, b, c, d, e, f, g, h\}. A federate subscribing to object class X can discover the Y instance as an X. Since attribute "h" is not present in instances of class X, that information is lost.

A federate can subscribe to multiple classes in a class hierarchy. If a federate subscribed to classes W and X, the following would be true (see Figure 5.7):

1. Instances of object class W would be seen without promotion.
2. Instances of object class X would be seen without promotion.
3. Instances of object class Y would be seen as instances of object class X.
4. Instances of object class Z would be seen as instances of object class W.

Some attributes may not have assigned values. It depends on what the originating federate has published for this object and the extent to which other federates have contributed to what’s known about the instance.
Interaction Subscription

- Each federate subscribes to the interaction classes it wishes to receive.
- It is not possible to subscribe to individual parameters of an interaction class.
- A federate is informed about a new interaction instance if:
  - The federate has subscribed to the interaction class of the instance
  - The instance can be promoted (i.e., up the hierarchy) to a subscribed interaction class.

When a federate discovers an object, it learns the object class of the instance. If the federate discovers the object instance to be of object class $X$, it will always believe the object’s type to be $X$. If a federate subscribes to class $X$ and not to class $Y$, it will discover $Y$ instances as $X$ instances. If the federate subsequently subscribes to class $Y$, object instances previously discovered as $X$ instances (via promotion) will continue to be seen as $X$ instances. Subsequently discovered instances of object class $Y$ will be discovered as instances of object class $Y$.  

5.3.4 Interaction Subscription

As with object classes, each federate subscribes to the interaction classes it wishes to receive (Figure 5.8). It is not possible to subscribe to individual parameters of an interaction class. Again, interactions are "all or nothing." As with object classes, a federate is informed about a new interaction instance if (a) the federate has subscribed to the interaction class of the instance or (b) the instance can be promoted (i.e., up the hierarchy) to a subscribed interaction class. When an interaction instance is promoted, only the parameters of the subscribed class are presented to the receiving federate.

---

Rediscovery of an object instance can be forced using the Local Delete Object Instance service. After object class $Y$ was subscribed to, a federate could "locally delete" all instances of object class $X$ to rediscover the objects based on the federate’s new subscriptions.
5.3.5 Control Signals

In Figure 5.5, above, Federate #1 indicated that it was capable of producing \( Y \) instances, but could only provide the attributes \{\( b, e, f, g, h \}\). In that same figure, Federate #2 subscribes to attributes \{\( a, b, c, d, e \}\) for object class \( X \). The \( Y \) instances produced by Federate #1 are discovered as \( X \) instances by Federate #2 (Figure 5.5).

Federate #2 is only interested in a few of the \( Y \) attributes produced by Federate #1. As discussed previously, Federate #2 cannot access attribute "h" since the attribute isn't a part of class \( X \). Further, Federate #2 has no interest in attributes \{\( f, g \}\). Of the information Federate #1 is able to produce \( Y:\{b, e, f, g, h\} \), only the information \( Y:\{b, e\} \) is required – assuming Federate #2 is the only other federate in the federation.

The RTI issues control signals to indicate the information Federate #1 should produce. By default, a federate should refrain from producing object updates unless the Local RTI Component (LRC) has indicated that a consumer exists. If Federate #1 is first on the scene (i.e., there are no consumers), it will never be signaled to begin registering \( Y \) instance information.

Once Federate #2 arrives, the LRC will indicate to Federate #1 that it should register any instances of object class \( Y \) with the federation execution and it should start providing updates for \( Y:\{b, e\} \). If Federate #2 goes away, Federate #1 will be told to stop registering instances of object class \( Y \) and to stop providing updates for \( Y:\{b, e\} \).

Each LRC informs its federate (via callbacks) which object attributes and which interactions to start or stop producing based on consumer demand. Each federate’s Simulation Object Model (SOM) will identify the extent to which the federate does or does not make use of the control signals provided by the LRC.

5.4 Object Publication and Subscription

Each federate is responsible for identifying its publication and subscription interests to the RTI LRC using the RTI_RTIambassador’s methods subscribeObjectClassAttributes() and publishObjectClassAttributes(). The interaction diagram shown in Figure 5.9, Object Publication and Subscription, illustrates the procedure for building the information required to use these methods.

The publish and subscribe methods both require an RTI_ObjectClassHandle and an RTI_AttributeHandleSet. The LRC has an internal (numeric) representation for object classes, object class attributes, interaction classes and interaction class parameter string representations that appear in the FDD file. RTI_RTIambassador’s methods such as getObjectClassHandle() and getAttributeHandle() translate character descriptions into
The (abstract) class `RTI_AttributeHandleSet` identifies a set of attributes – e.g., \{a, b, c, d\}. To express interest in publishing or subscribing to an object class, the following steps are required for RTI 1.3-NG.

For each object class to be published:

a) Obtain the handle for the current object class.

b) Create a free-store allocated `AttributeHandleSet` using the static `create()` method in the class `AttributeHandleSetFactory`.

c) For each attribute the federate can publish:
   i) Obtain the handle for the current attribute.
   ii) Add the handle to the `AttributeHandleSet`

d) Publish|Subscribe the `AttributeHandleSet` for the object class.

5.5 Throttling Publications

The LRC signals a federate to start or stop registering object instances for all published object classes and generating interactions for all published interaction classes.

5.6 Object Declaration

Excerpts from `HelloWorld`, illustrating publication and subscription using RTI-NG 1.3, are included in Part 5 of the `Lab Notebook`. General comments about the use of the RTI services are included below.
5.6.1 Dynamic Object Publication and Subscription

Each call to `publishObjectClassAttributes()` and `subscribeObjectClassAttributes()` for an object class replaces previous calls. The methods `unpublishObjectClass()` and `unsubscribeObjectClass()` should be called when a federate is no longer interested in any attributes of an object class.

5.7 Publishing and Subscribing Interactions

Registering publication and subscription interest in interaction classes is more straightforward than registering interest in object classes. Figure 5.10, Declaring Interactions, identifies `RTI_RTIambassador` declaration management methods for interactions. Interactions are "all or nothing." Unlike object registration, you cannot specify interest in particular interaction parameters.

As with object class declaration, interaction interest can be declared dynamically. Each call to `publishInteractionClass()` and `subscribeInteractionClass()` for an interaction class replaces previous calls. The methods `unpublishInteractionClass()` and `unsubscribeInteractionClass()` should be called when a federate is no longer interested in an interaction class.
5.8 Object Management

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support object management. Object management includes instance registration and instance updates on the object production side and instance discovery and reflection on the object consumer side. Object management also includes methods associated with sending and receiving interactions, controlling instance updates based on consumer demand, and other miscellaneous support functions.

To create an object, the federate must first have published that object class using the declaration management services of the RTI. The publishing federate then registers the object using the declaration management services. This results in a discover callback by the RTI to any subscribing federate. (See Figure 5.11.)

5.9 Registering, Discovering, and Deleting Object Instances

Figure 5.12 illustrates the interactions required to register and to discover object instances. The RTI_RTIambassador()’s method registerObjectInstance() informs the Local RTI Component that a new object instance has come into existence. The method requires the object class of the new object instance and an optional object name. The method returns an RTI_RTIObjectInstanceHandle which the LRC uses to identify the particular object instance.
Objects

- To create an object, the federate must have published that object class (declaration mgt.)
- The federate registers the new instance of the object
- To discover this object, other federates must have subscribed to that object class (declaration mgt.)
- Those federates then discover the object instance

Figure 5.11 Objects

Registration introduces an object instance to the federation. It does not, however, provide attribute values for the instance. That requires a second step.

Each and every object can be deleted by exactly one federate. Initially, the federate that creates (registers) an object has the privilege to delete the object. In Figure 5.12, the RTI_RTIambassador’s method deleteObjectInstance() is called to remove a registered object. The FederateAmbassador’s callback removeObjectInstance() informs federates that a previously discovered object no longer exists. The RTI_RTIambassador’s method localDeleteObjectInstance() effectively "undiscovers" an object instance. This method does not ensure the object will be permanently undiscovered. This service is intended to be used when a federate discovers an object as an instance of an object class but would like to subscribe to object classes that extend the discovered class and then rediscover the instance based on the new subscriptions. The object instance will be rediscovered upon the next updateAttributeValue() invocation that meets the receiving federate’s subscriptions.

† The topic Ownership Management, covered in Module 2, explores functions for giving away the privilege to delete as well as the right to update various attributes.
5.10 Updating and Reflecting Object Attributes

To update one or more attributes associated with a registered object instance, a federate must prepare an \textit{RTI\_AttributeHandleValueMap}. This is similar to the \textit{RTI\_AttributeHandleSet} discussed previously in Section 5.4, \textit{Object Publication and Subscription}. An \textit{AttributeHandleSet} (AHS) identifies a set of attributes. An \textit{AttributeHandleValueMap} (AHVM) identifies a set of attributes and their values. In RTI 1.3-NG, the static function \texttt{RTI::AttributeSetFactory::create()} is used to construct a free-store allocated AHVM instance.\(^5\) In Section 5.2, \textit{Object Hierarchies}, the notation \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\} was used to identify four attributes by name. The notation can be extended to accommodate attribute values – e.g., \{\texttt{a = 5, b = "Hello", c = 14.79821, d = -12}\}.

Attribute updates are provided for an object instance via the \texttt{RTI\_RTIambassador's} method \texttt{updateAttributeValue()}\(^5\). The method requires an \texttt{RTI\_ObjectInstanceHandle}, which the LRC uses to identify an object instance, an AHVM, and a descriptive character string (RTI\_User Supplied Tag).

\(^5\) AHVM is actually an abstract class; so, the factory function produces an AHVM descendant (implementation).
Figure 5.13, *Object Management Updates*, illustrates the interactions required to
discover and to reflect updates for external object instances. Discovery is the
counterpart to registration. Reflection is the counterpart to attribute updates. The
*FederateAmbassador’s* callback method *discoverObjectInstance()* informs the federate
that a new object instance has come into existence. The method provides an object
handle which will be used to identify the object for subsequent updates, etc. The
method also identifies the object class of the new object instance. It is extremely
important to note that the *RTI_ObjectInstanceHandle* is a local (numeric)
representation maintained by the LRC. The same object instance is typically known by
different handle values in each federate.4

### 5.11 Encoding and Object Update

When producing an AHVM, the federate is responsible for any data marshalling
(encoding). The LRC knows nothing about data content. It knows the names of object
classes, the names of attributes and the numeric handle representations for object
classes and attributes. The *HelloWorld* demonstrates how an AHVM is produced for the
*Country* class introduced in previous chapters. Data is encoded and the length of the
encoding is communicated to the LRC. Ultimately, the AHVM is bound to an object
instance handle in an *updateAttributeValues()* invocation.

### 5.13 Exchanging Interactions

Interactions are constructed in a similar fashion to the way attribute updates are
constructed. Recall that objects persist, interactions do not. Each interaction is
constructed, sent, and forgotten.8 Interaction recipients receive, decode, and apply the
interaction. Recall that, as for objects, to send an interaction, the federate must have
published that interaction class (per declaration management). Also, as shown in Figure
5.14, to receive an interaction, a federate must have subscribed to that interaction class
(per declaration management).

---

4For instance, in a particular implementation of the RTI, Federate #4 may know an object instance as 1278443. Federate #12 may
know the object instance as 956. Each federate’s LRC takes care of translating the local instance representation as required when
communicating with other federates.

7The AHVM actually consist of triples, not pairs. The triple is (1) the attribute handle, (2) the corresponding value and (3) the
length of the encoding.

8Interactions can be retracted. See the manual pages from the *Programmer’s Guide* for details.
Object Management Updates

- RTI reflectAttributeValue()
- RTI updateAttributeValue()
- RTI provideAttributeValueUpdate()
- RTI requestAttributeValueUpdate()
- RTI provideAttributeValueUpdate()
- RTI requestAttributeValueUpdate()

Figure 5.13 Object Management Updates

Interactions

- To send an interaction, the federate must have published that interaction class (per declaration management)
- To receive this interaction, other federates must have subscribed to that interaction class (per declaration management)

Figure 5.14 Interactions
Figure 5.15, lists the classes and methods involved in generating and consuming interactions.

![Diagram of Object Management Interactions]

**Object Management Interactions**

Figure 5.15 Exchanging Interactions

### 5.14 Additional Object Control

Object attribute updates and interactions are conveyed between federates using one of two data transportation schemes — "reliable" and "best effort". For objects, the transportation scheme is specified at the level of individual attributes. For interactions, the transportation scheme is specified at the interaction level (i.e., not the parameter level). By default, the transportation scheme is specified per object/attribute name and per interaction name in the *FOM Document Data* (FDD) file.

It is possible to change the transportation scheme dynamically for one or more attributes of a specific object *instance* using the *RTIambassador’s* method `changeAttributeTransportationType()`. It is possible to change the transportation scheme dynamically for interactions by *class name* using the *RTIambassador’s* method `changeInteractionTransportationType()`. Figure 5.16 illustrates these functions.
Figure 5.16 also shows two callback methods – `turnUpdatesOnForObjectInstance()` and `turnUpdatesOffForObjectInstance()`. These methods are used to inform a federate whether there is external interest in updates for specific attributes of specific object instances.\(^9\)

### 5.14.1 Attribute Management

A federate may have created and registered an aircraft. If one or more federates exist and are interested in updates for this particular object instance, the LRC would issue the `turnUpdatesOnForObjectInstance()` callback to specify the particular attributes for which updates should be generated. If at some point, there was no further external interest in the aircraft, the LRC would invoke `turnUpdatesOffForObjectInstance()` – informing the federate to cease updates for this particular object instance.

---

\(^9\) These functions are companions to the declaration management callback methods `startRegistrationForObjectClass()` and `stopRegistrationForObjectClass()` (see Part 4, Declaration Management).
The federate should presume that there is no external interest in an object unless or until `turnUpdatesOn...()` is issued. Calls to `turnUpdatesOn...()` and `turnUpdatesOff...()` are cumulative. Each call to `turnUpdatesOn...()` adds to the set of attributes that should be updated. Each call to `turnUpdatesOff...()` removes attributes from the set of attributes that should be updated.

5.14.2 Enable/Disable Attribute Management

It is possible to disable the `turnUpdatesOn...()` and `turnUpdatesOff...()` callbacks. The `RTIambassador` methods `enableAttributeRelevanceAdvisorySwitch()` and `disableAttributeRelevanceAdvisorySwitch()` can be used to specify whether per object instance control signals are generated or suppressed.\textsuperscript{16}

\textbf{Figure 5.17 Scope Interactions}

\textsuperscript{16} These methods are not shown in the accompanying interaction diagram.
5.14.3 Attribute Scopes

Prior to communicating attribute updates for a particular object instance, the LRC will (at the federate's discretion) provide the preliminary callback `attributesInScope()` announcing that subsequent attribute updates for the specified object and specified attribute set may be forthcoming. A subsequent `attributesOutOfScope()` callback would inform the federate that subsequent attribute updates for the specified object and specified attribute set would no longer be provided. These signals will be generated or suppressed based on the "attribute scope advisory switch" which is set by the `RTIambassador` methods `enableAttributeScopeAdvisorySwitch()` and `disableAttributeScopeAdvisorySwitch()`. Figure 5.17 provides an interaction diagram for these methods.

Assignment

1. Continue the effort from part 4 to create a new, passive federate, the `UNFederate`, that collects and displays data from any running `HelloWorld` federates.

2. The `HelloWorld` federate is much more complicated than absolutely necessary as it serves as a template and demonstration of the structure of much more complex simulations. Try to eliminate as many unnecessary functions from the `UNFederate` federate as you can to make it as simple as absolutely possible. Many of the callback routines in the `FederateAmbassador` may be eliminated by simply not performing the actions in the `UNFederate` federate that trigger unnecessary callbacks.

3. Using the `HLA Course Lab Notes, Chapter 5`, and the source files for `HelloWorld`, determine the data structures used in the `HelloWorld` federate to store the attributes of an object of Class `Country`, the locations in the program where class `country` is published and subscribed, and where `country` objects are registered and discovered. If more than one function is actually involved in fulfilling one of the above actions, describe the use of each function. Repeat the above for interactions and their parameters used by the `HelloWorld` federate.

4. Determine the mechanisms used by the `HelloWorld` federate to control excess message traffic and object registrations, and where the corresponding calls are located in the code.

5. Determine why there are so many variants of the update and constructor functions for Class `Country` in the `HelloWorld` federate, and the purpose of each variant.

Suggested Readings

Level Architecture (HLA) – Federate Interface Specification.

2. Read Chapters 7, and 8 of the *High Level Architecture Run-Time Infrastructure Programmer’s Guide*. Also look up the interface specifications for the interfaces used in this lesson in the Appendices to this Guide.