Early Experiences from Migrating to the HLA Evolved C++ and Java APIs

Björn Möller
Per-Philip Sollin
Mikael Karlsson
Fredrik Antelius

bjorn.moller@pitch.se
per-philip.sollin@pitch.se
mikael.karlsson@pitch.se
fredrik.antelius@pitch.se

Keywords:

HLA Evolved, FOM Modules, Web Services, Fault Tolerance, DLC, RTI

ABSTRACT: *Several previous papers have described a number of new concepts and services in the HLA Evolved (IEEE 1516-2009) standard, such as fault tolerance, update rate reduction and modular FOMs. This paper focuses on the particulars of the updated C++ and Java APIs. One reason for updating the APIs is to support the new or extended HLA functionality. Another reason is to make it easier to switch between different RTI implementations by simply replacing an RTI library file.*

This paper describes some early experiences from migrating from HLA 1.3 and HLA 1516-2000 to the new C++ and Java APIs for some commonly used HLA functionality. It is based on migration work done both for some basic federates as well as some general-purpose HLA tools.

Both the C++ and Java APIs have seen an evolution in the data types used for handles, which will affect every migration effort. For the C++ API there are also changes in the memory management schemes. The way that optional arguments are handled in the APIs has also evolved.

All federates will need to introduce the new connect/join sequence. Otherwise the majority of the HLA service functionality is very similar to earlier HLA versions or even simplified. Federates still using HLA 1.3 DDM may however need a major revision.

A few basic examples of using the new Encoding helpers are also given. This paper also shows how the new standardized time types are used.

Some properties of the HLA Evolved Dynamic Link Compatibility API (EDLC API) based on the earlier SISO DLC standard (SISO-STD-004.1-2004) are explained in detail. One particular feature here is the ability to dynamically choose between different RTI implementations at runtime. The relationship between the link compatibility, standardized time types and extendable transportation types is also explained.

This paper isn't a complete migration cookbook. Still it is intended to give developers some insights that are useful for planning their HLA Evolved migration efforts.

1. Introduction

The High-Level Architecture (HLA) [1] is a standard for simulation interoperability. There are several standards for exchanging data between computer applications but HLA addresses a number of requirements that are specific for simulation such as handling of logical (scenario) time and maintaining a large shared state.

There also exists an earlier standard for simulation interoperability called Distributed Interactive Simulation (DIS) [2]. Two important things must be noted when comparing HLA to DIS: the handling of domain models and the choice between protocol and services.

1.1 Where is the domain model?

To effectively exchange information for any domain you must use a well-defined information model. DIS has a fixed information model for defense platform simulations built into the standard. In HLA the user supplies the information model at runtime as an XML file with a flexible information model called the Federation Object Model (FOM). There are also pre-defined reference FOMs available that can be adapted and extended.

A fixed information model works well if all requirements for information exchange are met by this model. However, this demands that future requirements are predicted when the standard is defined. If a standardized model needs to be extended and/or adapted or if a different domain is simulated the flexible information model is better.

1.2 Protocol or services?

DIS is defined as a protocol where each participating application implements the functionality that they need to use. In HLA the standard describes a set of services that a Run-time Infrastructure (RTI) has to implement. These services are then accessed through local calls to a library (for C++ and Java) or using Web Services.

Simple operations, like updating the position of an aircraft on a best-effort basis, are easy to implement in each application. More advanced operations like reliably providing an interaction to a limited set of subscribing applications or coordinated time advance, are better provided as services. This helps avoiding extensive and error-prone re-implementations in several applications.

No matter if you are designing a protocol or an API great attention has to be paid to a number of factors such as parameters, data types, correct calling sequences, error handling and more.

1.3 A New Version of HLA

A new version of HLA, informally called "HLA Evolved" [3] is expected to be released during 2009. An overview of technical changes has been provided in an earlier paper [4]. HLA Evolved is the third major version of HLA. This paper describes some of the designs chosen for the HLA Evolved APIs. It also describes some early experiences from migrating to the new C++ and Java APIs.

2. Designing an API for HLA

This section gives a background on the design of an API based on the HLA Interface Specification with some notes on how these have evolved throughout the HLA versions.

HLA describes a set of services in a highly generalized format. It may at first seem trivial, or at least straightforward, to map these services to an API in a particular programming language. In practice a lot of additional design is needed and a large number of decisions need to be taken. There may also be several coding styles given a particular language.

HLA Evolved includes three APIs: C++, Java and Web Services [5]. There are big similarities between C++ and Java where service invocations are local calls within the same process to a dynamically linked library. These APIs can easily be called by programs written in the same language. They can usually also be called, with some effort, by applications written in other languages, given that there are mechanisms for making calls between the two languages.

When providing the same services through APIs in different languages, you always have to strike a balance between making the APIs as similar as possible and keeping each API true to the specific language. For example, Java provides automatic garbage collection while C++ uses explicit memory management. Trying to design APIs that hide that difference is bound to fail. There may also be technical limitations to take into account. A Web Services API must consider the fact that the Web Services will introduce a significant latency compared to the direct calls of Java and C++.

The Web Services API is fundamentally different since it describes a way of accessing the HLA services through http calls containing requests and responses structured using XML. There are no specific requirements on implementation languages on the RTI or federate side. It can in fact be argued that the Web Services API may be the best way to understand HLA because of the service-oriented focus. The federate and the RTI will typically also execute on different hosts or at least in different processes, as opposed to the C++ and Java APIs where they usually run in the same process.

2.1 Basic data types

When implementing the HLA services in an API it is necessary to describe exactly how each parameter type maps to a data type in the particular API. The WSDL API language specifics lists around 50 non-complex data types, such as Federation Execution Name (typically a Unicode String) and Dimension Bound (typically a 32 or 64 bit integer). Some of the data types are opaque, basically an array of bytes, for example a user supplied tag and are expected to be interpreted by the federate code. The main HLA standard text also uses concepts called designators and handles which are used to identify for example an interaction class or an object instance. In the API they will typically map to either a name in the form of a string or a language-specific object that the RTI can use to look up requested item.

There has been an evolution from the first HLA API to the HLA Evolved API where more modern programming practices have been introduced. It is beyond the scope of this paper to cover this in detail but the following example is given. In the API it is necessary to represent different kind of objects, such as attributes and object classes. These representations are often referred to as handles. In earlier versions of the API, integers were used for all different types of objects. This may at first seem to be a clear and simple solution. The problem here is that it is easy to use the wrong variable when calling a function, for example passing an integer representing an object class to a service expecting an attribute. This problem would usually not be detected until runtime, if at all. The solution is to introduce separate classes for different types of handles. This is a little bit more complex since it introduces new data types and new classes in the API. On the other hand it makes many mistakes obvious at compile-time rather than runtime. It is considerably less costly to discover a bug during development instead of integration or deployment.

2.2 Complex data types

The API also contains complex parameters such as lists of data structure, for example AttributeHandleValuePairSet. In this case different constructs are available for use in different languages. The Web Services solution would be a simple XML sequence. In Java from version 5 and up, there are typed collections, where it is possible to specify what kind of objects that a collection is supposed to hold. The typed collections provide compile-time checking of the objects that are put into a collection. In earlier Java versions, collections simply held a collection of objects which meant that the type checking wasn't done until runtime.

In the C++ API for HLA 1.3 [6], all container types, such as lists and maps, were custom solutions that were declared by the standard and had to be implemented by the RTI implementer. This was due to the fact that C++ provided no standard solutions at the time the HLA 1.3 standard was written. The next version, HLA IEEE 1516-2000, migrated to using collection types as defined by the Standard Template Library. This removed the burden of implementation from the RTI implementer. However, support for the Standard Template Library was very limited in the compilers of that time. The workaround used was to put duplicates of classes from the Standard Template Library [7] in the HLA IEEE 1516-2000 standard, thereby removing the requirement on the compiler. The Standard Template Library soon became a part of more or less all compilers, and the HLA standard took advantage of that by removing the duplicates and using the real classes.

2.3 Optional parameter handling

Many HLA services contain several optional parameters. A service with n optional parameters will in the general case result in 2^n possible combinations.

One example is the Reflect Attribute Values that has 5 optional parameters, such as Time Stamp and Producing Federate, resulting in 32 possible combinations of parameters, although some of these combinations would never occur.

The Java and the C++ APIs have different approaches to handling optional parameters. The C++ language has support for optional parameters where a default value can be given for use whenever a parameter is unspecified by the calling program. This was used to handle optional parameters in the C++ API. The Java language does not provide this mechanism. Instead, a combination of method overloading (different variants of a method with different parameter sets) and alternate methods (related methods with the same parameter set but different names, e.g. subscribeInteractionClass and subscribeInteractionClassPassively) was used.

Optional parameters have to be handled differently depending on whether the service is implemented by the RTI or by the federate. RTI-implemented services using the C++ style of optional parameters with default values can easily detect the default values and act accordingly. For federate-implemented services, such as Reflect Attribute Values, it was decided that default, or "empty", values was not appropriate for unavailable parameters. This left only the method of overloading which in the Reflect Attribute Values case produced eighteen different combinations of parameters. This was deemed unacceptable. The solution was to take some of the optional parameters and put them in a separate data structure call SupplementalReflectInfo with flags signaling whether the optional parameters were valid or not. This reduced the number of overloads from eighteen to three.

In Web Services the information that is passed in a call contains both the parameter name and the value. This makes the handling of optional parameters simple. Each call can be inspected to find out exactly which parameters that were sent.

2.4 Referencing "state"

The main text of the HLA standard says very little about referencing and maintaining the state of the federate and the RTI, or to be more exact, the Local RTI Component (LRC) of a particular federate. This can be seen as the state of the federate's session. As part of a session it is possible to create a Federation Execution and to join Federation Executions.

For C++ and Java there are two concepts: The RTI Ambassador, for which the federate maintains a reference (pointer) and the Federate Ambassador for which the RTI maintains a reference. A pair of connected ambassadors can be seen as a session that can then be used for example for creating and joining a Federation Execution.

An ambassador pair can only support one joined federate and a federate can only join one federation at a time. The RTI Ambassador will thus also maintain the state of the Federation on behalf of the federate.

For Web Services a corresponding session is maintained through a "cookie" which is the native way to maintain a reference to a session. Unfortunately Web Services offers several types of cookies. In this case a cookie for the http session was chosen to achieve the broadest compatibility with commonly used Web Services frameworks.

2.5 Calls and callbacks

The general text in the HLA specification says little about how the federate calls the RTI and even less on how callbacks are delivered to the federate. In C++ and Java RTI calls are implemented as method invocations on the RTI ambassador. Callbacks are implemented as method invocations on the Federate Ambassador that are carried out by the RTI. The Web Services API on the other hand has to rely on principles commonly available in Web Services frameworks. This means that the federate needs to "poll" the RTI for available updates.

Another question is at what point in time callbacks are to be delivered to a federate. In HLA 1.3, the solution introduced was to call a method on the RTI ambassador called "tick", later renamed to Evoke Multiple Callbacks. No callbacks would be delivered until they were Evoked from the RTI. Another implementation is the Immediate callback delivery where callbacks are delivered to the federate as soon as they are available to the LRC. This results in lower latency but also requires the application to be thread-safe, i.e. properly handle concurrency, since a callback could be delivered in a separate thread at any time.

2.6 Concurrency and reentrancy

In IEEE 1516-2000 and earlier versions, the question of concurrency and reentrancy was left untouched. This inevitably led to differences between the available RTI implementations. Some RTIs handled concurrent calls to the same RTIambassador instance by queuing them and executing them in order. Other RTIs simply threw an exception in this case. Making RTIambassador calls from within callbacks was another thing that was handled differently by different RTIs. HLA Evolved provides strict guidelines as to how an RTI should behave in this area.

2.7 Other aspects

The use of templates in the C++ API for IEEE 1516-2000 introduced a problem in that the implementation-specific parts that were used to instantiate the templates became part of the binary interface of the RTI. This meant that it was basically impossible to create link-compatible RTIs. The only way to accomplish that would be if all RTIs used the same implementation-specific parts, which was highly unlikely. This problem was solved by replacing the template-based solution with a solution based on the Pimpl idiom [8] which separates the public API parts from the implementation-specific parts.

The API defined in IEEE 1516-2000 lacked an explicit disconnect mechanism. In C++, this could be handled by making the deletion of the RTIambassador object perform the necessary cleanup. In Java, there is no delete call. Instead, the application simply releases its reference to the RTIambassador and lets the Garbage Collection mechanism do the cleanup. The problem is that the timing of the garbage collection is not deterministic. Adding a Disconnect service gave the federate the opportunity to signal that it had finished using the RTI and that a cleanup could be made. A corresponding Connect service helps to complete the handling of a federate's lifecycle. The Connect service also provides a standardized place for the federate to provide specific settings to the RTI.

User-defined time types were a feature of IEEE 1516-2000. However, the technical implementation was not fully defined. The mechanism for accessing different time-types was limited, both in Java and C++. This was fixed in HLA Evolved by the introduction of Time Factories. Another very useful thing was the addition of standardized time types with well-defined behavior. This will make porting of time-managed federates between different RTIs much simpler.

2.8 The DLC and the EDLC API

HLA Evolved incorporates a number of design changes that were introduced in the SISO DLC API for HLA 1516 [9][10]. This API had the same functionality as the official HLA IEEE 1516-2000 API but also included a number of improvements to make it easier to switch between different RTIs. The DLC API also removed the use of templates that prevented link-compatibility. An earlier paper [4] introduces the term "EDLC API" for the Evolved DLC API to make it easier to distinguish between the DLC and EDLC APIs.

The HLA Evolved EDLC APIs make it even easier to write RTI-independent federates. A number of additional issues for RTI plug-and-play have been resolved. The programmatic selection of Evoked versus Immediate callback delivery, as previously described, is one example. The standardized time types makes it possible for federate developers to rely on standardized, proven and readily available time representations in the RTI, independent of RTI supplier. Incompatible, non-standard transportation types is another issue which has been resolved by the introduction of a clear fallback scheme to standardized transportation types.

3. Migrating C++ Code to HLA Evolved

Some basic federates as well as some general tools have been migrated to HLA Evolved. The focus in this paper is to cover some general issues and experiences that most federate developers will encounter. Note that the code sections provided are just samples. This is not intended to be a complete migration cookbook. To make the differences as clear as possible a comparison is made with HLA

HLA 1.3 – C++ Startup Code

```
RTI::RTIambassador rtiAmbassador;
rtiAmbassador.createFederationExecution("myFederationName", "myFDD.fdd");

RTI::FederateHandle federateHandle =
    rtiAmbassador.joinFederationExecution("myFederateType",
                                          "myFederationName", myFedAmb);
```

HLA Evolved – C++ Startup Code

```
auto_ptr< RTIambassador > _rtiAmbassador;
RTIambassadorFactory* rtiAmbassadorFactory = new RTIambassadorFactory();

_rtiAmbassador = rtiAmbassadorFactory->createRTIambassador();
_rtiAmbassador->connect(myFedAmb, L"myRTI", HLA_IMMEDIATE);

vector<wstring> FOMmoduleFiles;
FOMmoduleFiles.push_back(L"myEvolvedFOM.xml");
_rtiAmbassador->createFederationExecution(
   L"myFederationName", FOMmoduleFiles, L"", L"HLAfloat64Time");

FederateHandle federateHandle = _rtiAmbassador->joinFederationExecution(
   L"myFederateName", L"myFederateType", L"myFederationName", FOMmoduleFiles);
```

Figure 1: C++ code for federate startup

1.3, released in 1998. Developers familiar with HLA 1516-2000 or the SISO DLC API may notice that a gradual development has taken place.

It should also be noted that from a functional perspective HLA Evolved is a superset of the previous HLA 1516-2000 standard. A developer knowledgeable about the previous standard only needs to understand the new concepts while the existing ones remain unchanged. Developers working with HLA 1.3 may want to read up on HLA 1516-2000. From a coding perspective however many parameters have changed and new data types have been introduced so existing code may require a lot of fairly straightforward updates.

In order to use HLA Evolved it is of course necessary to have a FOM that follows the new format. This migration can be done in minutes using a COTS tool if no further restructuring into FOM modules is desired. This assumes a HLA 1516-2000 FOM. The step from HLA 1.3 to the HLA 1516 level may require more work, for example adding data type information.

This section discusses C++ experiences based on code examples. Some experience with C++ is assumed. The Java oriented reader may want to skip this section and go directly to section 4. All exception handling has been removed in these samples to improve the clarity.

3.1 Connecting, Creating and Joining in C++

The first thing a federate needs to do is to create a federation execution and join the federation. Sample code for this is provided in Figure 1. Some interesting differences are highlighted in the code.

The RTI ambassador is maintained through the C++ auto_ptr construct, which is a class template available in the C++ Standard Library[7]. An auto_ptr ensures that the object to which it points gets destroyed automatically when control leaves a scope. An RTI ambassador is created using an RTIambassadorFactory.

Maybe the biggest difference between earlier versions and HLA Evolved is the separate Connect step. This allows the federate to try to connect to an RTI using a specific set of settings. The connect operation can of course fail and can be retried later or with alternate settings.

The settings for Connect are indicated in a string known as the Local Settings Designator, in this case “myRTI”. The exact interpretation of this parameter is done by each RTI implementation. This may be an RTI settings file, a label in a settings file or simply the name of the host that runs the RTI exec. If no settings are provided the default settings of the RTI implementation will be used. A federate developer is advised to provide some flexibility here by for example fetching this value from a configuration file or by interrogating the user.

Another interesting parameter is the callback mode which can be either HLA_IMMEDIATE or HLA_EVOKED. The former indicated that callbacks will be delivered in a separate thread as soon as they are available while the latter indicates that they will be delivered when the Evoke Multiple Callbacks or Evoke Callback service is called. The

HLA 1.3 – C++ Getting a Handle

```
ObjectClassHandle objectClassHandle =
   rtiAmbassador.getObjectClassHandle("Restaurant");
```

HLA Evolved – C++ Getting a Handle

```
ObjectClassHandle objectClassHandle =
   rtiAmbassador.getObjectClassHandle(L"Restaurant");
```

Figure 2: C++ code for handles

immediate mode has the advantage of guaranteeing the lowest latency. The Evoked mode has the advantage of not requiring thread-safe programming.

As the federate is now connected to an RTI it is time to create a federation execution. While earlier version of HLA required one monolithic FOM to create a federation execution, HLA Evolved enables you to provide the FOM as several FOM Modules [11]. All the predefined concepts such as standard data types and MOM data are provided automatically by the RTI in a FOM module called the HLAstandardMIM. In the code example we can see how a list of FOM modules is constructed and then provided upon creating a Federation Execution. The second parameter is an empty string indicating that the HLAstandardMIM should be used. It is also possible to provide a user-extended MIM at this point using the third parameter (an empty string in this example).

The last parameter indicates the time representation to be used if sending and receiving data with HLA timestamps and/or using full time management. There are two standardized representations that are always available in all RTI implementations called the “HLAfloat64Time” and “HLAinteger64Time”.

The Join Federation Execution call now contains an argument for supplying a federate name. It is also possible to provide additional FOM modules when joining. In this case the same list is provided just to illustrate the principle. There are two slight differences in what FOM data that can be provided between the two calls Create Federation Execution and Join Federation Execution. The MIM (standard or extended) can only be provided in the Create Federation Execution. It is also necessary to provide at least one FOM module when creating federation execution to provide the Switches table. All other FOM modules should either provide an identical Switches table or no Switches table at all.

Before leaving this code example it should be pointed out that this is the code where it is most likely that an exception will be thrown. The RTI may not be available as expected, the connection parameters may not be valid in the current environment, the FOM may be incorrect, etc.

3.2 Getting Handles in C++

Handles are used in many places. The first place that it is used in the federate code is usually when publishing and subscribing, which would be the next thing to do. The code snippet in Figure 2 shows how to get handles.

HLA 1.3 – C++ Receive Interaction

```
void MyFederateAmbassador::receiveInteraction (
        RTI::InteractionClassHandle       theInteraction,
  const RTI::ParameterHandleValuePairSet& theParameters,
  const RTI::FedTime&                     theTime,
  const char                              *theTag,
        RTI::EventRetractionHandle        theHandle)
```

HLA Evolved – C++ Receive Interaction

```
void MyFederateAmbassador::receiveInteraction (
   InteractionClassHandle theInteraction,
   ParameterHandleValueMap const & theParameterValues,
   VariableLengthData const & theUserSuppliedTag,
   OrderType sentOrder,
   TransportationType theType,
   LogicalTime const & theTime,
   OrderType receivedOrder,
   SupplementalReceiveInfo theReceiveInfo)
```

Figure 3: C++ code for interactions

Encoding in C++

```
ParameterHandleValueMap parameters;

wstring ws(L"Hello there");
HLAunicodeString unicodeMessage(ws);
parameters[_parameterText] = unicodeMessage.encode();

VariableLengthData userSuppliedTag;
_rtiAmbassador->sendInteraction(_messageClass, parameters, userSuppliedTag);
```

Decoding in C++

```
HLAunicodeString messageDecoder;
messageDecoder.decode(theParameterValues[_parameterText]);
wstring ws = messageDecoder;
```

Figure 4: C++ code for encoding and decoding in HLA Evolved

This short code snippet contains very few changes. The only obvious difference is that a Unicode string is used as an argument for getting a handle. However, behind the scene in the standard header files the HLA 1.3 ObjectClassHandle is really a typedef for a C++ "unsigned long". In practice an AttributeClassHandle, which is also an "unsigned long" would work just as well, opening up for a plethora of coding mistakes. The HLA Evolved header files defines a separate class for each type of handle, making these coding mistakes obvious already when compiling.

3.3 Object Management in C++

There are several minor differences in the HLA Evolved object management when compared to HLA 1.3. Most of them appeared already in HLA 1516-2000, in particular the ability to do asynchronous registration of object instance names, which have now been enhanced to register multiple names in one call.

One particular change in several calls is that a number of optional parameters have been put in a "supplemental" information block. Figure 3 shows the Receive Interaction callback. In this case a pointer to a Supplemental Receive Info object is provided. This object may or may not contain the optional parameters SentRegion and producingFederate. The latter is a new parameter that can be used to determine what federate that sent a particular interaction. Another very useful change is that object names can be reused.

3.4 Encoding and Decoding data in C++

HLA evolved introduces a set of classes [12] that makes it easier to encode/decode your application data into the format that is used when it is sent and received in the HLA services. The exact format is specified in a FOM according to the OMT format.

Sending time-stamped interactions

```
HLAfloat64Time timestamp(17.0);

_rtiAmbassador.sendInteraction(
   _messageClass, parameters, userSuppliedTag, timestamp);
```

Receiving time-stamped interactions

```
void MyFederateAmbassador::receiveInteraction (
   InteractionClassHandle theInteraction,
   ...
   LogicalTime const & theTime,
   ...
{
   ...
   wstring ws = messageDecoder;
   HLAfloat64Time const & floatTime =
      dynamic_cast<HLAfloat64Time const &>(theTime);
   double d = floatTime;
```

Figure 5: C++ code using a standardized time representation in HLA Evolved

It is vital that this format is followed to achieve any degree of interoperability. The following example (figure 4) shows how to take a C++ wstring, convert it into a HLAunicodeString and then send an interaction. It also shows how to decode the corresponding string when received from another federate.

3.5 Standardized time types in C++

HLA Evolved provides two standardized time representations HLAfloat64Time and HLAinteger64Time. The name of the required time representation is provided as part of the Create Federation Execution call, as can be seen in figure 1. The following sample (Figure 5) shows how to send a time-stamped interaction for time=17. It also shows how the interaction is received and the time-stamp is retrieved.

Note that any custom time representation will need to be adjusted to fit into the new API.

3.6 More practical experiences

The code snippets above have been extracted from two migration efforts that are described below.

A basic federate was migrated by a developer with HLA 1516 experience but no previous knowledge about the HLA Evolved APIs. The federate contained object registration, removal and attribute updates and request/provide. It also included sending and receiving of interactions. This effort required the developer to study the standard, the APIs and some papers for three days. The porting then took approximately two days including testing. None of the new HLA Evolved features were introduced and the federation agreement didn't change. The HLA 1516-2000 FOM was migrated using an OMT Tool (Pitch Visual OMT 1516 version 1.5) in minutes using the default settings.

Another case is a set of general HLA tools at Pitch that separates the code that calls the RTI from the rest of the application. The developer was familiar with HLA Evolved but the functionality of the code was somewhat larger and the requirements for performance and reliability were higher. This case required approximately three days of work. It should also be noted that these migrations were performed against a preliminary version of the HLA Evolved APIs. The above code samples have later been revised to match the expected final version of the APIs as of January 2009. Some additional code samples are available in a separate document [13].

4. Migrating Java Code to HLA Evolved

Some basic federates as well as some general tools have been migrated to HLA Evolved. This section discusses Java code and repeats some remarks that were already given in the C++ section.

HLA 1.3 – Java Startup Code

```
RTIambassador rtiAmbassador =
   RTI.getRTIambassador(rtiHost, CRC_PORT);

URL myFddUrl = new URL("file://./myFDD.fdd");

rtiAmbassador.createFederationExecution("myFederationName", myFddUrl);

MobileFederateServices timeServices =
   new MobileFederateServices(
       new LogicalTimeDoubleFactory(),
       new LogicalTimeIntervalDoubleFactory());
int federateHandle = rtiAmbassador.joinFederationExecution(
   "myFederateType", "myFederationName", myFedAmb, timeServices);
```

HLA Evolved – Java Startup Code

```
RtiFactory rtiFactory = RtiFactoryFactory.getRtiFactory();
RTIambassador rtiAmbassador = rtiFactory.getRtiAmbassador();

rtiAmbassador.connect(
   myFedAmb, "myRTI", CallbackModel.HLA_IMMEDIATE);

URL[] fomModules = new URL[] {new URL("file://./myEvolvedFOM.xml")};

rtiAmbassador.createFederationExecution(
   "myFederationName", fomModules, "HLAfloat64Time");
FederateHandle rtiAmbassador.joinFederationExecution(
   "myFederateName", "myFederateType", "myFederationName", fomModules);
```

Figure 6: Java code for federate startup

HLA 1.3 – Java Getting a Handle

```
int objectClassHandle =
   rtiAmbassador.getObjectClassHandle("Restaurant");
```

HLA Evolved – Java Getting a Handle

```
ObjectClassHandle objectClassHandle =
   rtiAmbassador.getObjectClassHandle("Restaurant");
```

Figure 7: Java code for handles

4.1 Connecting, Creating and Joining in Java

The first thing a federate needs to do is to create a federation execution and join the federation. Sample code for this is provided in Figure 6. Some interesting differences are highlighted in the code

An RTI ambassador is created using an RtiFactory-Factory. In a more advanced case it would be possible to have several RTIs installed on a computer and select which one to use based on this Factory mechanism.

Maybe the biggest difference between earlier versions and HLA Evolved is the separate Connect step. This allows the federate to try to connect to an RTI using a specific set of settings. The connect operation can of course fail and can be retried later or with alternate settings.

The settings for Connect are indicated in a string known as the Local Settings Designator, in this case “myRTI”. The exact interpretation of this parameter is done by each RTI implementation. This may be an RTI settings file, a label in a settings file or simply the name of the host that runs the RTI exec. If no settings are provided the default settings of the RTI implementation will be used. A federate developer is advised to provide some flexibility here by for example fetching this value from a configuration file or by interrogating the user.

Another interesting parameter is the callback mode which can be either HLA_IMMEDIATE or HLA_EVOKED. The former indicated that callbacks will be delivered in a separate thread as soon as they are available while the latter indicates that they will be delivered when the Evoke Multiple Callbacks or Evoke Callback service is called. The immediate mode has the advantage of guaranteeing the lowest latency. The Evoked mode has the advantage of not requiring thread-safe programming.

As the federate is now connected to an RTI it is time to create a federation execution. While earlier version of HLA required a monolithic FOM to create a federation execution HLA Evolved enables you to provide the FOM as several FOM Modules [11]. All the predefined concepts such as standard data types and MOM data are provided automatically by the RTI in a module called the HLAstandardMIM. In the code example we can see how a list of FOM modules is constructed and then provided upon creating a Federation Execution. The parameter indicating if a non-standard MIM shall be used is omitted in this example, which means that the HLAstandardMIM will be used.

The last parameter indicates the time representation to be used if sending and receiving data with HLA timestamps and/or using full time management. There are two standardized representations that are always available in all RTI implementations called the “HLAfloat64Time” and “HLAinteger64Time”.

The Join Federation Execution call now contains an argument for supplying a federate name. It is also possible to provide additional FOM modules when joining. In this case the same list is provided just to illustrate the principle. There are two slight differences in what FOM data that can be provided between the two calls Create Federation Execution and Join Federation Execution. The MIM (standard or extended) can only be provided in the Create Federation Execution. It is also necessary to provide at least one FOM module when creating federation execution to provide the Switches table. All other FOM modules should either provide an identical Switches table or no Switches table at all.

Before leaving this code example it should be pointed out that this is the code where it is most likely that an exception will be thrown. The RTI may not be available as expected, the connection parameters may not be valid in the current environment, the FOM may be incorrect, etc.

4.2 Getting Handles in Java

Handles are used in many places. The first place that it is used in the federate code is usually when publishing and subscribing, which would be the next thing to do. The code snippet in Figure 7 shows how to get handles.

This short code snippet contains only one change. The type of the objectClassHandle variable is now a class instead of an integer.

4.3 Object Management in Java

There are several minor differences in the HLA Evolved object management when compared to HLA 1.3. Most of them appeared already in HLA 1516-2000, in particular the ability to do asynchro-

HLA 1.3 – Java Receive Interaction

```
void receiveInteraction (
   int                 interactionClass,
   ReceivedInteraction theInteraction,
   byte[]              userSuppliedTag,
   LogicalTime         theTime,
   EventRetractionHandle eventRetractionHandle)
```

HLA Evolved – Java Receive Interaction

```
void receiveInteraction(
   InteractionClassHandle interactionClass,
   ParameterHandleValueMap theParameters,
   byte[] userSuppliedTag,
   OrderType sentOrdering,
   TransportationTypeHandle theTransport,
   LogicalTime theTime,
   OrderType receivedOrdering,
   SupplementalReceiveInfo receiveInfo)
```

Figure 8: Java code for interactions

nous registration of object instance names, which have now been enhanced to register multiple names in one call. Another very useful change is that object names can be reused.

One particular change in several calls is that a number of optional parameters have been put in a "supplemental" information block. Figure 8 shows the Receive Interaction callback. In this case a pointer to a Supplemental Receive Info object is provided. This object may or may not contain the optional parameters SentRegion and producingFederate. The latter is a new parameter that can be used to determine what federate that sent a particular interaction.

4.4 Encoding and Decoding data in Java

HLA evolved introduces a set of classes [12] that makes it easier to encode/decode your application data into the format that is used when it is sent and received in the HLA services. The exact format is specified in a FOM according to the OMT format. It is vital that this format is followed to achieve any degree of interoperability. The following example (Figure 9) shows how to take a Java String, convert it into a HLAunicodeString and then send an interaction. It also shows how to decode the corresponding string when received from another federate.

4.5 Standardized time types in Java

HLA Evolved provides two standardized time representations HLAfloat64Time and HLAinteger64Time. The name of the required time representation is provided as part of the Create Federation Execution call, as can be seen in figure x. The following sample (Figure 10) shows how to send a time-stamped interaction for time=17. It also shows

Encoding in Java

```
ParameterHandleValueMap parameters =
   _rtiAmbassador.getParameterHandleValueMapFactory().create(1);
HLAunicodeString messageEncoder = _encoderFactory.createHLAunicodeString();
messageEncoder.setValue(message);
parameters.put(_parameterText, messageEncoder.toByteArray());

byte[] userSuppliedTag = null;
_rtiAmbassador.sendInteraction(_messageClass, parameters, userSuppliedTag);
```

Decoding in Java

```
HLAunicodeString messageDecoder = _encoderFactory.createHLAunicodeString();
messageDecoder.decode(theParameters.get(_parameterIdText));
String message = messageDecoder.getValue();
```

Figure 9: Java code for encoding and decoding in HLA Evolved

Sending time-stamped interactions

```
HLAfloat64Time timestamp =  hlaFloat64TimeFactory.makeTime(17.0);

byte[] userSuppliedTag = null;
_rtiAmbassador.sendInteraction(
   _messageClass, parameters, userSuppliedTag, timestamp);
```

Receiving time-stamped interactions

```
void receiveInteraction(
   InteractionClassHandle interactionClass,
   ...
   LogicalTime theTime,
   ...
{
   ...
   String message = messageDecoder.getValue();
   HLAfloat64Time floatTime = (HLAfloat64Time)theTime;
   double d = floatTime.getValue();
```

Figure 10: Java code using a standardized time representation in HLA Evolved

how the interaction is received and the time-stamp is retrieved.

Note that any custom time representation will need to be adjusted to fit into the new API.

4.6 More practical experiences

The code snippets above have been extracted from two migration efforts that have been described in section 3.6. The same developers did the Java migration, which means that the time to study the standard was shared between these efforts.

The basic federate was migrated to Java in less than one day. The standard tools were migrated to Java in 2 days including testing.

Some additional code samples are available in a separate document [14].

5. Discussion

This section summarizes some thoughts and additional observations that were made when migrating federates to HLA Evolved.

One update that was necessary for all federates was to call the Connect call before creating or joining a federation execution. The federate ambassador reference that was previously provided as part of the Join call had to be moved to the Connect call. Note that the Connect call is also where all federates need to decide if callbacks shall be delivered during an Evoke call or delivered immediately in a separate thread.

The use of supplemental information data structures significantly reduced the implementation effort. It reduces the number of places where code needs to be duplicated. It also reduces the risk of implementing the "wrong" version of the callback method, potentially resulting in loss of callbacks.

The addition of standardized time representations as well as the way to specify which time representation to use must be considered a major enhancement. In earlier HLA versions each federate could specify which time representation to use implicitly (based on which dynamic link library that happened to be on the path) or in the join call from each federate, opening up for a mismatch between federates. The time representation is now provided as part of the Create Federation Execution call, establishing a common time representation for all participating federates.

Some of the tools that we migrated from earlier HLA version repeatedly called ("polled") some HLA services to detect if it was still connected to a federation. These calls could now be removed since loss of connection is clearly signaled by the RTI in HLA Evolved. Note however that only a limited level of fault tolerance was implemented.

HLA Evolved uses a new OMT format and also offers the ability to provide a FOM as modules. Maybe the biggest migration effort for tools that read FOMs (i.e. generic data loggers) is to implement parsing and merging of FOM modules, an effort that took more than a month.

In general, HLA Evolved really makes it possible to write more advanced tools by introducing more flexibility, discoverability and transparency. It is possible to enumerate which federations that currently exists for a given RTI. The new "federate name" property makes it easy recognize federates. FOM modules contributed from different federates can be enumerated, retrieved and inspected. New aspects like "producing federates" can be conveyed for any interaction or attribute update received. The use of well-known, pre-defined time representations makes it easier for general tools to inspect and use time stamp values.

6. Conclusions

A number of C++ and Java federates, both specialized federates and more general tool, have been migrated to HLA Evolved. In general the effort has been very limited, basically days or a few weeks, with the exception of tools that explicitly parse and process FOM data.

The general process for migrating a federate to HLA Evolved can be described as:

1. Decide if the federation is to use any of the new HLA Evolved features, for example fault tolerance, update rate reduction or standardized time types.
2. Migrate the FOM to the HLA Evolved, for example using COTS tools
3. Migrate the federate code to HLA Evolved, potentially implementing any new HLA Evolved functionality used and possibly adding Encoding Helpers.
4. Test each federate and the federation.

In this case the main new functionality added in step 1 was some fault tolerance. The FOM conversion was done using a COTS tool in less than an hour.

For the migration we noted that only one substantial change was required throughout all federates: the addition of the Connect/Disconnect calls. We also found the encoding helpers very helpful for several data types and would definitely recommend them as a way to save implementation and debugging time. If the federation uses time management then a choice must be made which standardized time type in HLA Evolved to use. If the federation used a custom time type that cannot be replaced with one of the standardized time types then the custom time type must be migrated to HLA Evolved.

To summarize, migrating federates to HLA Evolved has generally been simple requiring only a few days for simple federates and a few weeks for more advanced federates. Developers that want to take advantage of some of the new features, like fault tolerance, smart update reduction or even FOM processing, will need to plan for some additional time.

7. Federate and Tools List

The paper builds upon migration experiences from the following federates and tools:

- Sample chat federate (C++)
- Sample chat federate (Java)
- Tiny "Air Traffic Control" federation (C++)
- Tiny "Air Traffic Control" federation (Java)
- COTS Data Logger ("Pitch Recorder")
- COTS Federation Manager ("Pitch Commander")
- COTS Visualizer ("Pitch Google Earth Adapter")
- COTS Middleware Code Generator ("Pitch Developer Studio")
- A small proprietary CGF (Java)

The following tools were used during the migration:

- Pitch pRTI v4.0 prerelease 1 and 2
- Pitch Visual OMT version 1.5.0 and 1.5.1

The above tools support HLA Evolved draft 4 (April 2008) and draft 5 (February 2009) respectively.

References

[1] IEEE: "IEEE 1516, High Level Architecture (HLA)", www.ieee.org, March 2001.

[2] IEEE: "IEEE 1278.1, IEEE Standard for Distributed Interactive Simulation - Application Protocols", www.ieee.org, May 1993

[3] Roy Scrudder, Gary M. Lightner, Robert Lutz, Randy Saunders, Reed Little, Katherine L. Morse, Björn Möller. "Evolving the High Level Architecture for Modeling and Simulation". Proceedings of the 2005 Interservice/ Industry Training, Simulation & Education Conference, Paper No. 2157, National Training Systems Association, December 2005.

[4] Björn Möller, Katherine L Morse, Mike Lightner, Reed Little, Robert Lutz. HLA Evolved – A Summary of Major Technical Improvements, Proceedings of 2008 Spring Simulation Interoperability Workshop, 08F-SIW-064, Simulation Interoperability Standards Organization, September 2008

[5] Björn Möller, Staffan Löf. "A Management Overview of the HLA Evolved Web Service API", Proceedings of 2006 Fall Simulation Interoperability Workshop, 06F-SIW-024, Simulation Interoperability Standards Organization, September 2006.

[6] "High Level Architecture Version 1.3", DMSO, www.dmso.mil, April 1998

[7] Alexander Stepanov and Meng Lee "The Standard Template Library". HP Laboratories Technical Report 95-11(R.1), November 14, 1995. (Revised version of A. A. Stepanov and M. Lee: "The Standard Template Library", Technical Report X3J16/94-0095, WG21/ N0482, ISO Programming Language C++ Project, May 1994.)

[8] "Exceptional C++", Herb Sutter, Addison-Wesley Professional, ISBN 978-0201615623

[9] SISO: "Dynamic Link Compatible HLA API Standard for the HLA Interface Specification" (IEEE 1516.1 Version), (SISO-STD-004.1-2004)

[10] Len Granowetter, "Design of the Dynamic-Link-Compatible C++ RTI API for IEEE 1516", Proceedings of 2004 Fall Simulation Interoperability Workshop, .04F-SIW-086, Simulation Interoperability Standards Organization, September 2004

[11] Björn Möller, Björn Löfstrand, Mikael Karlsson. "An Overview of the HLA Evolved Modular FOMs", Proceedings of 2007 Spring Simulation Interoperability Workshop, 07S-SIW-108, Simulation Interoperability Standards Organization, March 2007.

[12] Björn Möller, Mikael Karlsson, Björn Löfstrand. "Reducing Integration Time and Risk with the HLA Evolved Encoding Helpers". Proceedings of 2006 Spring Simulation Interoperability Workshop, 06S-SIW-042, Simulation Interoperability Standards Organization, April 2006.

[13] Pitch Technologies. "Migrating a C++ Federate to HLA Evolved", www.pitch.se/hlaevolved

[14] Pitch Technologies. "Migrating a Java Federate to HLA Evolved", www.pitch.se/hlaevolved

Author Biographies

BJÖRN MÖLLER is the vice president and co-founder of Pitch, the leading supplier of tools for HLA 1516 and HLA 1.3. He leads the strategic development of Pitch HLA products. He serves on several HLA standards and working groups and has a wide international contact network in simulation interoperability. He has twenty years of experience in high-tech R&D companies, with an international profile in areas such as modeling and simulation, artificial intelligence and Web-based collaboration. Björn Möller holds an MSc in computer science and technology after studies at Linköping University, Sweden, and Imperial College, London. He is currently serving as the vice chairman of the SISO HLA Evolved Product Development Group.

PER-PHILIP SOLLIN is a developer and consultant at Pitch. He studied Computer Science at Chalmers University, Sweden.

MIKAEL KARLSSON is the Chief Architect at Pitch overseeing the world's first certified HLA IEEE 1516 RTI as well as the first certified commercial RTI for HLA 1.3. He has more than ten years of experience of developing simulation infrastructures based on HLA as well as earlier standards. He also serves on several HLA standards and working groups. He studied Computer Science at Linköping University, Sweden.

FREDRIK ANTELIUS is a Lead Developer at Pitch and is a major contributor to several commercial HLA products. He holds an MSc in computer science and technology from Linköping University, Sweden.