Object-Oriented HLA - Does One Size Fit All?

Björn Möller
Fredrik Antelius

bjorn.moller@pitch.se
fredrik.antelius@pitch.se

Keywords:

HLA, Middleware, C++, Java, FOM, Code generation, OO-HLA

ABSTRACT: *The HLA RTI is accessed using a standard service API that is independent of application domain. A popular approach to simplify the use of HLA is to hand-code, or to generate an object-oriented RTI middleware with an API that closely matches a specific FOM. This is informally known as Object Oriented HLA (OO-HLA).*

This paper describes OO-HLA with pros and cons and points to some important design considerations. It also summarizes some practical experiences from designing, implementing and using a COTS product that generates OO-HLA middleware in C++ and Java.

OO-HLA middleware can greatly simplify the implementation of HLA interfaces for federates, improve quality and save time and money. At the same time, such a FOM-specific API will never be able to support generic, domain-independent tools, for example for federation management and data logging, thus limiting the potential for reuse. Another fact that reduces the potential of OO-HLA APIs, as compared to the current standardized HLA API, is that one single FOM will never be able to support all current and future interoperability needs.

There are fundamental differences between object oriented programming languages and HLA. A number of assumptions about how a federate wants to use for example ownership, DDM and time management must be made in order to support these services in an object oriented API. Similarly it is also necessary to make a number of assumptions about the HLA-based interplay between federates in order to fully use object oriented features such as method invocations.

The overall conclusions are as follows:

- It is of great benefit to both have access to the traditional, generic HLA API and to be able to hand-code or generate FOM-specific object-oriented middleware.

- A commonly used subset of the full HLA functionality directly matches the object-oriented constructs.

- For more advanced HLA concepts the object oriented paradigm is too limited to allow a direct mapping. These concepts, like time management, ownership and DDM can indeed be made available based on additional utility classes, design patterns and exception handling. There are several potential structures of such an API, for example with respect to time-stamping and ownership. Different designs will match different users needs.

- It is possible to create one standardized API pattern for OO-HLA but it is more likely that several different designs patterns are necessary to support different users needs.

1. Introduction

The High-level Architecture (HLA) [1][2][3] was originally developed by the US DoD as a successor to both DIS [4], that supports real-time platform simulations, and ALSP [5], that supports event-driven theater-level simulations. HLA is the leading, and actually the only standard that fully supports interoperability for any information exchange model between real-time simulations (like Live simulators), paced real-time systems (like Virtual simulators) and time-stepped and event-driven systems (like Constructive simulations).

There are great benefits from using exactly one interoperability standard for connecting all different kinds simulators. This facilitates reuse across organizations and enables the development of common knowledge, tools, components and processes.

At the same time, just like an advanced sports car may introduce challenges to a less experienced driver, all the functionality and flexibility of the

HLA standard and API may present a challenge to a new developer. A popular way to circumvent this is to provide middleware to simplify the use of HLA.

A number of such middleware implementations for the HLA have been produced during the last decade. A few of them have been commercial products whereas most of them have been in-house efforts in industry or government projects. In many cases they have attempted to match HLA objects to object-oriented programming objects in C++ or Java.

The authors of this paper have been lead designers of a middleware generator (Pitch Developer Studio [6]) that generates both C++ and Java source code. This paper summarizes our analysis of important aspects of object-oriented HLA (OO-HLA) and describes some practical experiences and design aspects in section 4.

In a recent SISO initiative a study group for object oriented HLA (OO-HLA [7]) has also been suggested. The purpose of this paper is to shed some light on some object-oriented approaches and challenges for HLA middleware, to describe some practical experiences, and to give the author's views on the road ahead.

2.1 Interoperability and object oriented middleware

The discussion about object oriented middleware is in no way unique for HLA. This approach has been used for example for DIS as well as many non-standard (proprietary) interoperability approaches, both service-oriented and protocol-based. It shall also be noted at this point that the HLA API is already object-oriented using the main object classes RTIambassador for calls to the RTI and FederateAmbassador for callbacks from the RTI to the federate.

This makes sense since HLA is an interoperability architecture between systems (called federates), not necessarily between specific object instances in different systems. As an example, what is represented as a Brigade object instance in one system may be represented as a Brigade or, alternatively, as numerous Soldier object instances in another system.

For an object-oriented developer who is used to being in control of all objects in participating systems this is often perceived as a limitation or even a challenge. However, for many real life applications, where simulations are acquired from different suppliers, systems need to be reused and older systems are gradually replaced with newer systems, this is instead a must.

2.2 Flexible or Fixed FOM?

The representation of the information exchange model, here called the FOM (using HLA terminology), is crucial for the design of object oriented middleware. The information model may be either **fixed**, like in DIS or if the effort is limited to a particular FOM (like RPR2 [8]). It may also be **flexible** which means that it is supplied for example in a file (like the HLA FOM) or as part of the programming calls.

For a fixed information model it is possible to design an object oriented API that is also fixed. For flexible information models it is necessary to design a mapping whereby the API is derived for example from the FOM. If complex data types, like records or arrays, are described as part of the FOM this also requires a mapping.

For a fixed information model the most obvious implementation approach is to simply hand-code it. For a flexible information model a code generator may be the most efficient approach. If the fixed information has a lot of repeated items it makes sense to use a generator here too. In most cases where code generation is used, large static code chunks may still be hand-coded and independent of the FOM. These are sometimes put in a runtime library.

2. About HLA middleware

This sections examines some common types of middleware for HLA. The reader is assumed to have some knowledge of HLA.

2.1 Calling the RTI without middleware

An application that doesn't use any middleware will need to call the RTI directly using the standard services in the HLA Interfaces Specification, as shown in Figure 1.

The application instantiates an RTI ambassador to which it makes calls. Callbacks from the RTI is delivered to a Federate ambassador object that was initially supplied by the application. The developer needs to understand the required calling patterns, for example Creating a Federation, Joining a Federation, Publishing and Subscribing and then Registering an object. In addition to this, and just as

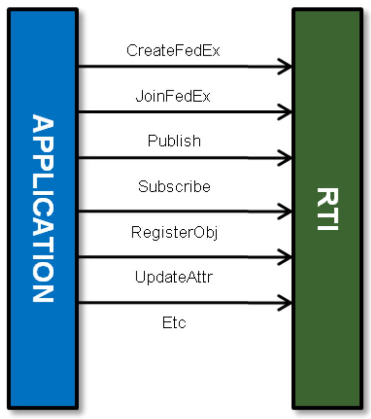


Figure 1: An Application without Middleware

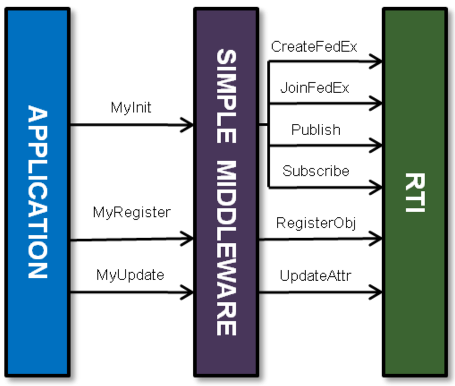


Figure 2: Simple HLA Middleware

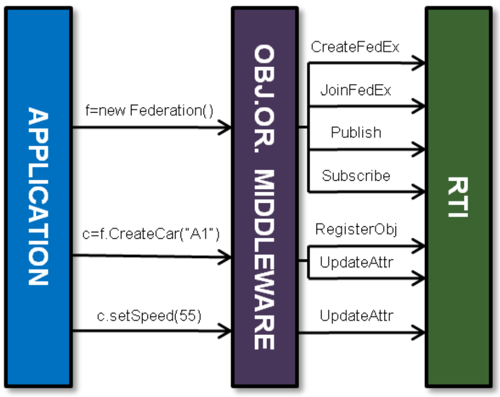


Figure 3: Object Oriented HLA Middleware

critical, the developer needs to develop code that handles the incoming attribute and parameter data, provided as byte arrays, and convert them to native variable values.

In practice almost all developers implementing their first federate start with an existing sample federate and extends it to fit their needs.

2.2 Simple HLA middleware

Many developer groups quickly find out that large pieces of code are repeated between different federate implementation projects. This usually leads to the development of simpler middleware libraries that abstracts and encapsulates commonly used HLA functionality.

In the example shown in Figure 2 all of the initialization steps, like Creating and Joining, are encapsulated by the MyInit call.

Most of the interplay with the RTI is still visible to the application and needs to be explicitly handled. It is possible to design this type of middleware in such a way that class and attribute names are provided as parameters to the middleware. This will make the middleware “FOM flexible”. A resulting drawback is that this degree of interpretation makes the implementation and use of the middleware error prone.

2.3 Object-oriented HLA middleware

Since HLA supports shared object instances across a federation, the next obvious step is to represent shared object instances as local C++ or Java objects. This can be seen as using the Proxy programming pattern.

Figure 3 shows an example of the use of an OO-HLA middleware. A local object of the class Car with the name “A1” is created. The speed attribute is later updated to 55.

Each shared HLA object instance is represented as an object oriented class instance. Locally created object oriented instances are automatically registered in the HLA federation. HLA object instances, created by other federates, sometimes known as *remote objects*, are automatically created as object oriented instances locally in the application.

When an attribute value is updated on a local object the middleware automatically sends an HLA update to the federation. When an HLA update is received the corresponding proxy object is updated, enabling the application to read the value whenever needed.

Figure 4 further illustrates how a *local* object in Federate A corresponds to a *remote* object in Federate B and vice versa. This mapping between the HLA architecture and object-oriented representation has many inconsistencies. It is really only the attribute of an object instance that a federate owns that can be seen as *local*. The ownership can also change over time. We have left the closed world of the object-oriented application behind and objects and attributes are now distributed across a federation. As attribute updates are sent over the RTI, corresponding *remote* objects will temporarily have different state.

It shall be noted that the mapping of HLA interactions is even less obvious. There are actually quite a few areas where the mapping between HLA and object oriented programming is less obvious and requires many additional assumptions, as will be shown later in this paper.

The use of the word *proxy* above may be questioned since it implies that there is an original

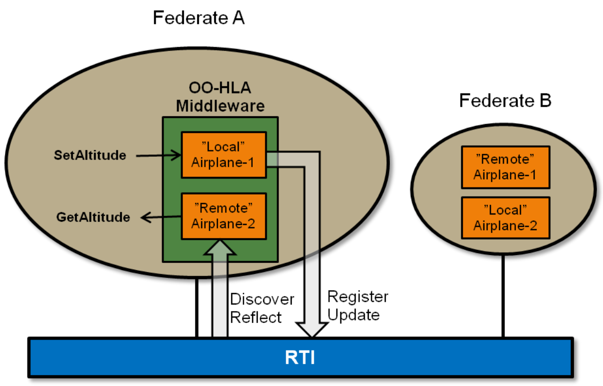


Figure 4: Objects and OO-HLA middleware

server object available in some particular application. The attributes of the proxy object may actually be owned by several different federates but seen from the local federate it may well be perceived as a proxy object.

2.4 Pros and cons of OO-HLA

The first and most obvious advantage of OO-HLA middleware is that it drastically reduces the learning curve for HLA. The developer can work with well-known concepts like instantiating objects and setting and getting member variables.

For an integration manager there are two obvious benefits: implementation time and quality. Simulators can be adapted to use HLA within a shorter and more predictable time frame. The integration events will also need less time since the common errors in encoding and decoding of data is less likely to occur.

Middleware can also provide best-practice patterns automatically, for example support for late joiners or fault tolerance. It is also possible to capture some aspects of federation agreements in the middleware.

There are also drawbacks or at least risks with using middleware. It may be tempting to configure the middleware to subscribe to and maintain more remote objects and attributes than necessary, which will limit the performance and scalability. The implementation may also provide more features than necessary, also resulting in reduced performance.

The middleware may in some cases even prevent implementation of the required HLA functionality, since hiding and grouping HLA service calls also gives the developer less control over them.

If the middleware is hard-wired to a specific FOM is impossible to implement certain types of general-purpose tools, like FOM-independent data loggers.

2.5 Mixing middleware and direct RTI calls

In theory it is possible to allow an application to call the RTI using both an OO-HLA API and the standard HLA API. This requires that there is no relationship or unwanted side effects of the two types of calls, otherwise the middleware's assumptions about the RTI state will fail. In practice there are usually many such relationships and side effects. More or less all HLA services (except maybe synchronization points) may at times be related. For example, it is unacceptable if an application directly calls functions like Time Advance Request, Unconditional Attribute Ownership Divestiture or Resign Federation Execution behind the scenes when the OO-HLA middleware is about to send an attribute update with a particular time stamp. This would make the federate break the HLA rules (and trigger an RTI exception).

2.6 More about HLA middleware

It is also possible to create middleware that can interoperate using several different standards or methods, for example using HLA or DIS.

Middleware may offer some degree of FOM Agility, which is the ability of an application to adapt to different FOMs. This agility will be limited by the information that is exchanged between the application and the middleware which means that it will mostly be of syntactic nature. If, for example, an application provides aircraft type, marking, nationality and geocentric coordinates to the middleware, it is possible to easily adapt to a FOM with a Lat/Long coordinate system using FOM agility functionality in the middleware. However, it will not be possible to publish the damage state without modifying the application.

Finally it shall be noted that other types of HLA middleware, in addition to the three types above, are also possible.

3. A Closer Look at OO-HLA

To fully understand OO-HLA it is necessary to understand some basic differences and to study how OO and HLA functionality can be mapped to each other.

3.1 Some fundamental differences

There are a few fundamental differences between an C++ or Java environment and an HLA Federation that needs to be understood:

Closed world assumption: The object oriented world is known in advance by the developer and can be fully understood and controlled. The federation on the other hand, including participating systems and their behavior, may not usually be fully understood by the developer and may vary from time to time.

Differences in life cycle: The life cycle of the federation may be different from the life cycle of the application. A reasonably fault tolerant federate may lose and then regains the connection to the federation.

Availability of objects: A traditional program, if correctly written, may be in full control of the life cycle of an object such as an aircraft. In HLA objects can either be locally created or created in remote applications and discovered locally. In an HLA federation on the other hand a reflected, “remote”, object may unexpectedly come and go.

3.2 Mapping OO and HLA functionality

Figure 5 shows an Aircraft object oriented instance and a corresponding HLA Aircraft object. Each of them follows the corresponding OO or HLA semantics.

Some part of OO and HLA, like the concept of object classes with attributes map very well. In order

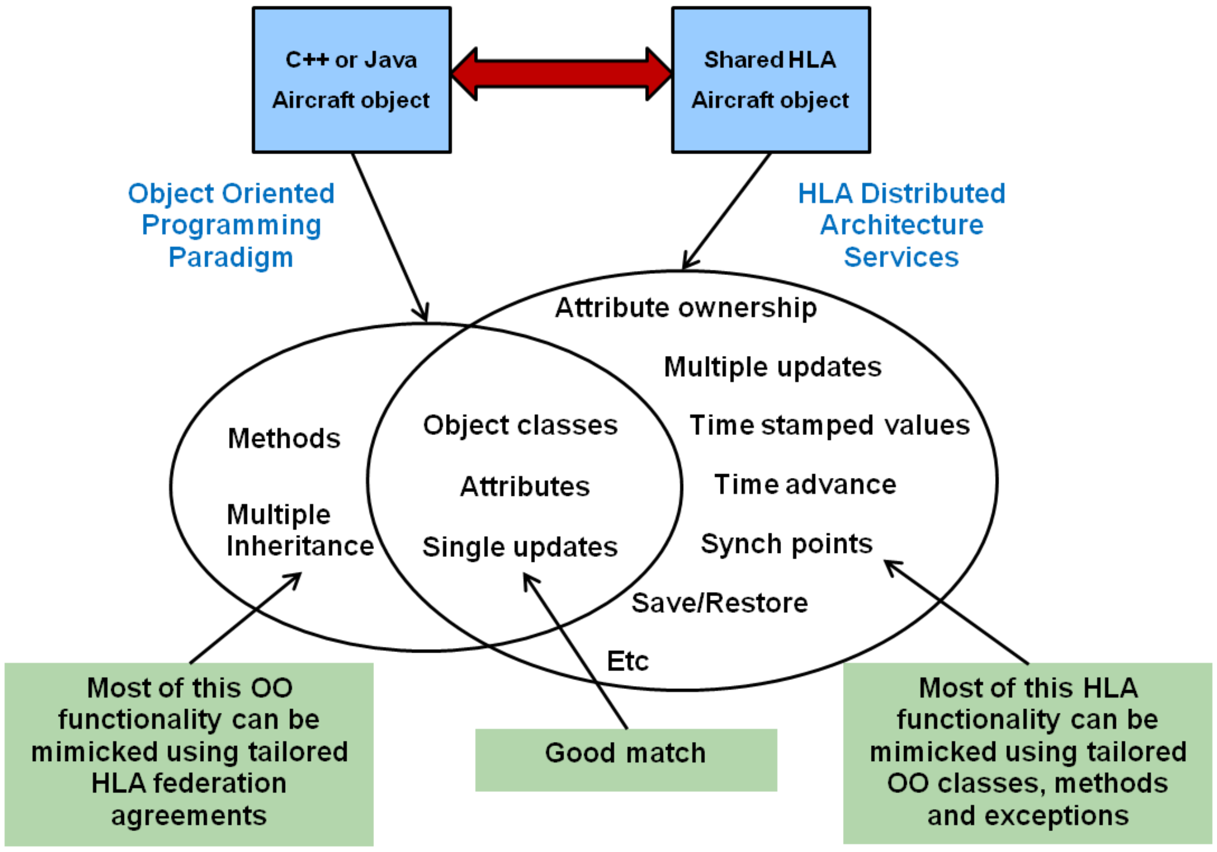

C++ or Java
Aircraft object
Shared HLA
Aircraft object
Object Oriented Programming Paradigm
HLA Distributed Architecture Services
Attribute ownership
Multiple updates
Methods
Object classes
Time stamped values
Attributes
Time advance
Multiple Inheritance
Single updates
Synch points
Save/Restore
Etc
Most of this OO functionality can be mimicked using tailored HLA federation agreements
Good match
Most of this HLA functionality can be mimicked using tailored OO classes, methods and exceptions

Figure 5: Providing an HLA API using OO and Vice Versa

to provide other HLA functionality, like attribute ownership, time stamped values and synchronization points it is necessary to make certain assumptions and/or add an extra layer of design in the object oriented API. Vice versa it may also be necessary to make certain assumptions in order to implement object oriented programming across the HLA services. Figure 6 provides a summary of the mapping. Green cells in the table indicates that a good match exists. Yellow cells indicate that it is necessary to make additional decisions and constructs in order to call HLA using OO middleware or to provide OO functionality across HLA.

3.3 Good matches

The following concepts have a good match between OO and HLA:

Object classes with subclasses. Both OO and HLA provides these constructs with similar semantics.

Class attributes. The semantics and structure of OO and HLA class attributes, including their inheritance is very similar.

Updating of single attributes. In this case HLA has object model a richer semantics but the basic semantics is similar.

3.4 Mimicking HLA semantics in OO

In the following cases a number of HLA concepts need to be mimicked using tailored OO classes or additional methods and exceptions:

Object world life span: An OO application with its locally instantiated objects is always online and available whereas the federation and its objects isn't available until the create/join/publish/ subscribe sequence has been carried out. Meta data, functionality and exceptions need to be introduced to handle this.

Synchronization points and Save/restore: Handlers, handshaking and exceptions need to be introduced.

Declaration of interest (publish/subscribe): Functionality for expressing interest in selected classes need to be introduced.

Object instance life cycle: Functionality for handling remote objects that may come and go needs to be introduced. It may be necessary to hide newly discovered objects to the application until certain, required attributes have been initialized.

Grouped attribute updates: In many cases several attribute values need to be updated as one atomic transaction which requires additional methods or classes.

HLA	Object Oriented
Life span/availability federation including fault tolerance. (Connect/Join/Resign/Disconnect/Fault handling)	Need to add meta data or functionality for checking availability or "on-line status" of federation and shared objects. May include error signalling.
Synch Points	Need to design Synch point handlers and corresponding application logic
Save/Restore	Need to implement state save and design handshaking.
Declaration of interest in objects and interactions (Pub/Sub)	Need to choose which objects and interactions to share.
Shared object instances (discover/remove)	Need to handle unpredictable life span of remote objects.
Object Classes with subclasses	Object Classes with subclasses
Need to make assumptions on how an interaction should be dispatched to an object instance on subscribing federate.	Method invocation on object instance
Class attributes	Class attributes
Updating of single attribute	Updating of single attribute
Need to make assumption on how key attributes map.	Object references using pointers
Grouped attribute updates	Need to create method for "atomic" update
Ownership of attributes	Need to create meta functions, handle ownership negotiation, handle lack of ownership, etc
Time Stamped attribute values	Need to add meta data for attributes with time stamp or use federate-wide time stamp
Time advance request/grant.	Need to add meta data to represent current time and OO calls to invoke time advance and callbacks for granting. Need to prevent updating during TAG.
Attribute value retraction	Need to provide functionality for monitoring and propagating retracted values
DDM filtering	Need to decide how application data maps to DDM regions for updates as well as for subscriptions.

Figure 6: Detailed Comparison of OO and HLA Semantics

Ownership of attributes: An OOHLA middleware needs to provide "meta-functions" for the OO attributes to initiate ownership transfer, to set the "transferable/acceptable" state, to determine current ownership status and to perform ownership negotiation when required. It may also be required to provide notification functionality for changes in ownership. Since there are several ownership transfer patterns an OOHLA middleware may only support a subset of these.

There are some "best-practices" that may be supported, for example sending a last update of an attribute value before ownership is released. This enabled the acquiring federate to pick up using the most recent attribute value.

In addition to this it is necessary to handle the situation where the application, through the OO middleware API, tries to update an attribute that is unowned by the federate. This may be considered an exception, an inconvenience or it may simply be ignored. An update of several attributes may run into the situation where only a subset of these attributes can be updated. This may be considered unacceptable since this was an atomic transaction or the issue may be ignored.

Time stamped attribute values and interacitons: HLA offers the ability to exchange time stamped attribute updates and interactions. Some applications may want to use application-wide time stamping, for example in frame-based (time-stepped) simulations. In other cases each shared attribute value or interaction may be associated with its own time stamp, for example in many event-driven simulations.

Time advance request/grant: This HLA functionality must be provided in an OO API. It may be used with or without the above time stamps. If both are used the middleware needs to manage the production of time-stamped data during time advance grants.

Attribute value retraction: Handlers for propagation the effect of retractions needs to be introduced

DDM Filtering: General functionality for specifying how DDM regions are derived from application date needs to be introduced, both for subscriptions and registering and updating object instances.

3.5 Mimicking OO semantics in HLA

In the following case OO concepts need to be mimicked using tailored federation agreements:

Methods: A federation agreement needs to be designed that describes how an HLA interaction should be mapped or resolved to a particular object class instance, for example using a "target object" parameter.

Pointers: In case OO objects use pointers to reference each other it is necessary to design a federation agreement that describes how they can reference each other using a globally valid unique identifier, like a name string.

Multiple inheritance: HLA only provides single inheritance. This isn't a problem since the OO classes in OO HLA are derived from the HLA FOM so we derive a class structure that may or may not have multiple inheritance from an OO class structure which always is limited to single inheritance.

3.6 Additional concerns when designing OO HLA

There are several additional design decisions that need to be made when creating an OO-HLA middleware:

Statefulness: It is of great benefit to have a stateful middleware that maintains full copies of the most recent published and subscribed attribute values. This will facilitate the support for late join in the federation. At the same time this limits performance and scalability. Maintaining a list of sent interactions for a federate that is temporary disconnected from the federation is just as useful in the short run as impossible during longer disconnections.

Data type handling: HLA attributes and attribute values can be mapped to C++ or Java attributes. The data type of the HLA attribute needs to be mapped to a corresponding native data type. This may be simple for some data types but more difficult for others. It may be necessary to create new C++ or Java data types for more complex values such as records or arrays.

An HLA 1516-2000 or HLA Evolved FOM contains a complete and unambiguous specification of how attribute, parameter and other federation data shall be encoded and decoded when transmitted (whereas the HLA 1.3 FOM doesn't).

Polling or notification of state changes: The application can gain insight into changes in attribute values for example by polling or by getting notifications from the middleware.

Compile time or runtime control: Many of the configuration aspects, like what to publish and subscribe or how to resolve interactions to object instances may be configured during generate/compile time or at runtime.

Handling of threading and mutability: The API can be implemented to support multi-threading or to use the thread of the application (using "tick" style calls). It may be wise to perform defensive copying of objects created by the middleware. This approach may differ between a Java API and a C++ API.

Support for specific federation agreements: Some federation agreements may require specific behaviors from the middleware. A simple example is to auto-achieve synch points. One example is the RPR FOM which uses the "periodic" update method, as opposed to the more obvious "on change" update criteria.

4. Experiences from Developing an OO-HLA Code Generator

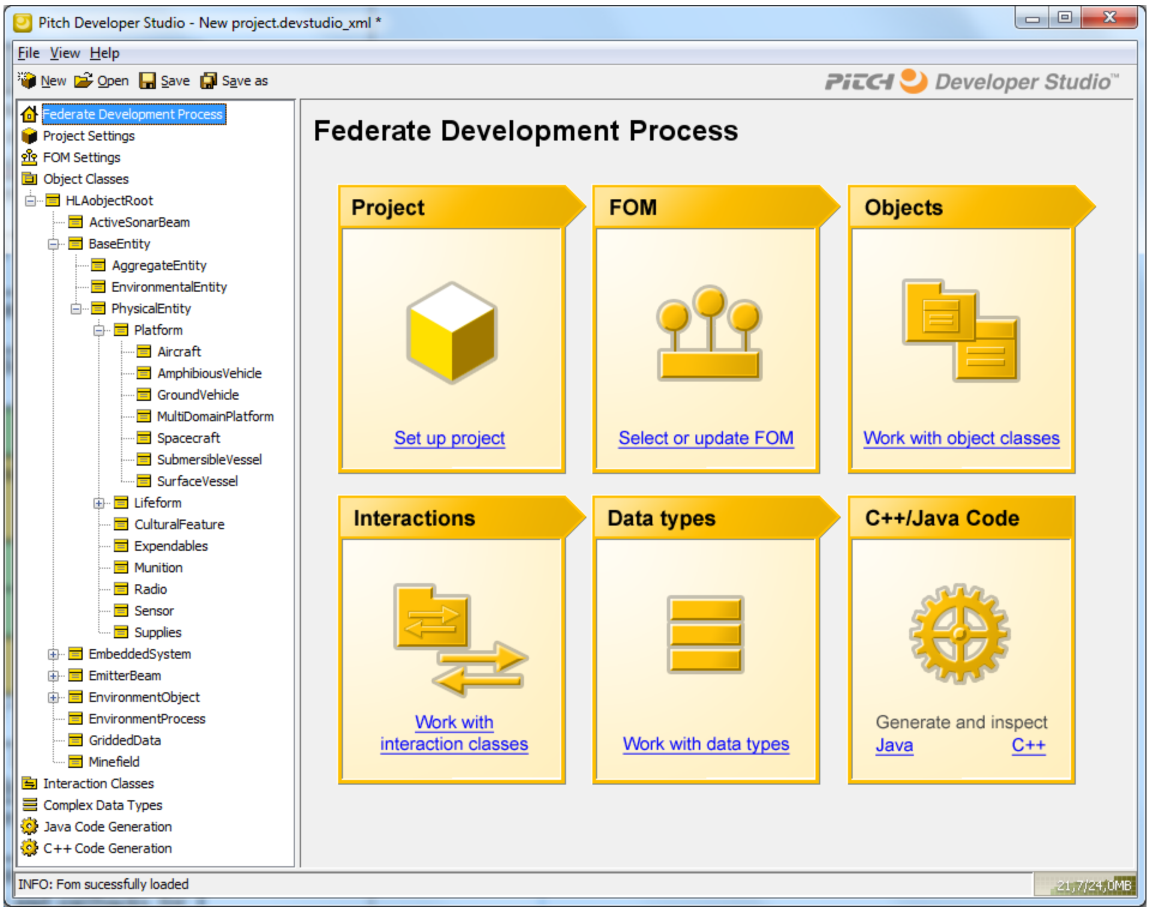
A COTS product (called Pitch Developer Studio) that generates OO-HLA middleware has been developed. It generates C++ code for 32 and 64 bit applications mainly on Windows and Linux as well as Java code for Java 5 and higher.

The work flow of the product is shown in Figure 7. In the first step the user provides some general settings, like application name, use of time management, use of MOM data, etc. In the second step the user selects a FOM and then selects and configures classes, attributes, interactions and data types. The FOM may later be replaced, which typically happens in a federation where the Federation Agreement and the FOM is extended. Finally code can be generated in C++ and/or Java for a number of compiler versions, including the generation of make or ant files as well as documentation.

4.1 Important Design Decisions

The overall goal of the product was to make it considerably easier and less costly and error-prone to adapt a simulator to HLA. The first version supports federation management (except save/restore), declaration management and object management. A second version, supporting selected aspects of ownership and time management is underway and will be released during 2010. A future version will support DDM.

While the creation of correct code templates for a



INFO: Fom sucessfully loaded

Figure 7: Sample OO-HLA Code Generator Workflow

code generator is a tedious task the most demanding task may well have been to create a design to meet this goal. Our analysis showed that generating code that supports all possible ways to utilize the HLA functionality would result in an API that was even more complex than the original HLA API. The following design decision were taken:

- Real-time, paced real-time and frame based simulation should all be supported. Event-driven simulations introduces considerably more complex handling of time-stamped data and was omitted.
- The general style of the API should be based on design patterns [9] such as observer/observable.
- The middleware should be stateful, saving the most recent value for each attribute value. This facilitates fault tolerance and late joiners.
- Convenience functions, like lookup tables for key attribute values and the dispatching of interactions as method calls should be added.
- Certain federation agreement aspects of the RPR FOM should be supported, like RPR non-standard data types, grouping of attributes in updates and convenience functions for spatial data.
- Round-trip support, where the user can generate new versions of the middleware is supported by providing an API that enables the user to maintain the middleware code separately from his own code. The generated code for a particular user configuration will maintain the same entry points and parameters.

The product is commercially available and it is currently in use by customers in Europe, North America and Asia.

5. Discussion – Does One Size Fit All?

In our initial designs of OO-HLA the generated code was mainly a function of the FOM and the HLA standard. As we kept extending the design we noted that other requirements, including the support for specific federation agreements and need for convenience functions, was just as important factors for the code generation. Our current work shows that the support for all aspects of HLA, for example both frame-based and event-based time management, would make the resulting code utterly complex.

Figure 8 shows our current understanding. To get useful middleware it's not enough to just generate code from the FOM. It is just as important to take into account for example:

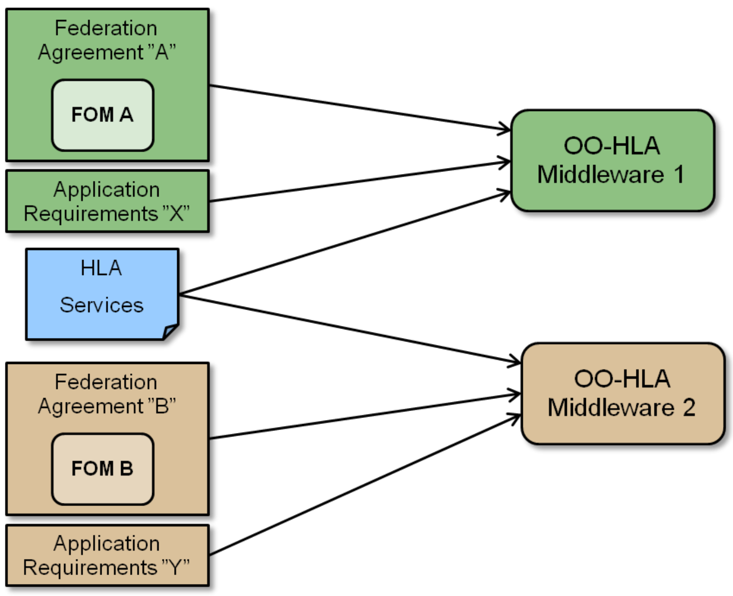


Figure 8: Federation Agreements and Requirements Affects the Optimal OO-HLA Design

Federation agreements, like how and when attributes are updated, how late joiners are handled, which type of overall time management that is used and more.

Application requirements, like the need for convenience functionality, dispatching of interactions, how and when the application learns about updated values, requirements for performance and scalability, multi-threading and memory management and more.

5.1 Examples with Conflicting Requirements

The following examples show some cases where we have found it hard to design one general OO-HLA API that meet the requirements and at the same time are practical to work with:

Non-standard update modes, like periodic updates or grouped updates where you always want to send updates of a group of attributes although only one attribute was changed. A well-known example is the widely used RPR FOM. It is of course possible to create good OO-HLA middleware for the RPR FOM but it would have many behaviors that would be unacceptable in other types of federations.

Different use of time management where there is a large number of simulations that would benefit from frame-based OO-HLA middleware. In this case the entire state of the objects in the OO-HLA middleware moves in the same time step, or frame. This middleware would be unacceptable for an event-driven simulation which can produce future attribute values with arbitrary time stamps. Both the use of time management services and the use of federate-wide versus attribute-specific time-stamps would need to be different.

5.2 Completeness or easy to use?

It is only for a subset of the HLA services that the object oriented paradigm makes things easier. For other functionality, like ownership management, the full set of HLA services still needs to be provided. The HLA standard lists 18 ownerships services which would all need to be supported for each attribute or groups of attributes from the same class. This makes the list of services very long without making HLA ownership any easier to use.

Another approach would be to have middleware that supports federation agreements with specific usage patterns. One example is a usage pattern where ownership is transferred between named federates. The middleware services would then be:

1. Transfer ownership to federate with name=X
2. Accept ownership transfer from federate with name=Y
3. Decline ownership transfer from federate with name=Y

Adding a number of patterns like this makes the middleware more useful and easily understood by federate developers. At the same time this makes the middleware even more complex if many different patterns would need to be supported.

We think it is a better idea to develop object oriented HLA middleware with a limited functionality that is highly useful to a particular class of federations than to create a "one size fits all" middleware with a full expansion of all possible usage patterns of everything in the FOM.

6. Conclusions

HLA is in essence a Service-Oriented architecture for distributed systems and Object Orientation is a programming paradigm, having fundamentally different object concepts. Still it is possible to create middleware that allows a user to interoperate using HLA through an object-oriented API for a specific FOM. In this case the middleware is designed so that the object-oriented classes match the shared object classes. This enables convenient access to a subset of the HLA functionality.

This type of middleware can make it easier, quicker and less error-prone to adapt simulations with similar requirements to the HLA standard. The result is that it can extend the market for HLA by enabling more users to use HLA in an easy way.

Each specific OO-HLA middleware will be different depending on the federation agreement (including the FOM) that needs to be supported as well as additional application requirements. Any attempt to fully capture all HLA services and all application requirements and commonly used federation agreements in an OO-HLA API will result in an API that is considerably more complex than the HLA specification, forfeiting its original purpose. OO-HLA APIs can complement, but not replace the traditional way to access HLA, using the standard HLA API.

Finally two things should be noted:

- A move towards a service oriented approach for distributed systems requires more than just the replication of attribute values.
- OO-HLA simplifies the access to commonly used HLA services but doesn't necessarily simplify the design of distributed systems.

References

[1] "High Level Architecture Version 1.3", DMSO, www.dmso.mil

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

[3] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, A.k.a. "HLA Evolved"

[4] IEEE: "IEEE 1278, Distributed Interactive Simulation (DIS)", www.ieee.org,

[5] Weatherly, R.M., Wilson, A.L. and Griffin, S.P.: "ALSP - Theory, Experience, and Future Directions,", Proceedings of the 1993 Winter Simulation Conference, pp. 1068-1072, Los Angeles, CA, 12–15 December.

[6] Pitch Technologies: "Pitch Developer Studio User's Guide", October 2009

[7] Object-Oriented HLA Study Group reflector, SISO, www.siostds.org

[8] SISO, "Real-time Platform Reference Federation Object Model 2.0 ", SISO-STD-001 SISO, draft 17

[9] Gamma, E et.al. "Design Patterns: Elements of Reusable Object-Oriented Software", ISBN 0-201-63361-2

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.

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.