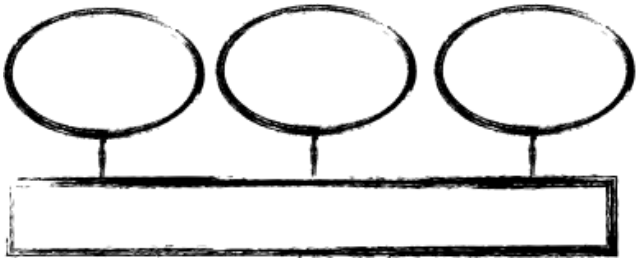


THE HLA TUTORIAL

A PRACTICAL GUIDE FOR DEVELOPING DISTRIBUTED SIMULATIONS

GET AN OVERVIEW OF HLA

LEARN HOW TO DESIGN HLA FEDERATIONS

UNDERSTAND HOW TO DEVELOP FEDERATION OBJECT MODELS

MASTER THE SERVICES OF THE RUN-TIME INFRASTRUCTURE

PART 1

STAY UP TO DATE!

THE MOST RECENT VERSION OF THIS DOCUMENT, CODE SAMPLES AND SOFTWARE ARE AVAILABLE AT

WWW.PITCH.SE/HLATUTORIAL

ABOUT THIS TUTORIAL

"The High-Level Architecture is a powerful technology for developing distributed simulations. It is amazing to see how people use it to make simulations work together in many different domains, to create simulations that have never been created before.

I have been involved in the HLA community since the mid 90's both in the standards development and the user community. I have also given at least one hundred HLA courses and seminars on four continents. With this document I want to offer everyone a practical introduction to HLA, based on these experiences.

This tutorial is mainly targeted at developers. I have written the tutorial as a small story where you can follow how an HLA federation is developed, step-by-step. It is intended to complement the HLA standard, which provides the final and complete specification of HLA.

I would also like to encourage the reader to become part of the HLA community through SISO, colleagues, numerous simulation conferences, papers, and of course the Internet. Enjoy the reading!"

Björn Möller

The team who developed "The HLA Tutorial" and the "HLA Evolved Starter Kit":

Björn Möller	Mikael Karlsson	Björn Löfstrand
Åsa Wihlborg	Steve Eriksson	Martin Johansson
Filip Klasson	Patrik Svensson	Pär Aktanius

Table of Contents

1. Introduction

- This tutorial provides both overview and technical details
- The purpose of HLA is interoperability and reuse
- HLA enables simulations to interoperate in a federation of systems
- HLA originated in the defense sector but is today increasingly used in civilian applications
- The HLA standard is part of policies, for example in NATO
- HLA enables a marketplace for simulations

1.1. How to use this tutorial

Welcome to the HLA Tutorial. This tutorial covers everything from an overview of what HLA is used for, to the practical design and development of HLA based systems. Here is a short guide for the reader:

- If you want a quick overview of what HLA is all about, you can start with reading this first chapter.
- If you want to understand the technical architecture of HLA, also read chapter two and three about the architecture and services of HLA.
- If you intend to understand how to develop HLA based systems then continue with the rest of the tutorial.

This tutorial is available as a stand-alone document or as part of the HLA Evolved Starter Kit. This kit contains software and samples with source code that you can run on your own computer, study, experiment with, extend or even use as a starting point for your own HLA development. This document contains instructions about practical labs that you can perform for many of the chapters.

This document is based on the latest version of HLA, commonly called HLA Evolved with the formal name IEEE 1516-2010. HLA Evolved builds upon earlier HLA versions such as HLA 1516-2000 and HLA 1.3.

1.2. HLA is about Interoperability and Reuse

The High-Level Architecture, HLA, is an architecture that enables several simulation systems to work together. This is called interoperability, a term that covers more than just sending and receiving data. The systems need to work together in such a way that they can achieve an overarching goal by exchanging services.

Why do we need to connect several simulation systems? One reason is that an organization may already have a number of simulation systems that need to be used together with newly acquired simulations or simulations from other organizations. Another reason is that there may be a requirement to simulate a "bigger picture", where models from different organizations interact. Experts from different fields need to contribute different models. In many cases it would also be a monumental task to build one big system that covers everything compared to connecting several different simulations.

One important principle of HLA is to create a **federation** of systems. Most simulation systems are highly useful on their own. With HLA, we can combine them to simulate more complex scenarios and chains of events. HLA enables us to reuse different systems in new combinations.

The two main merits of HLA are thus interoperability and reuse.

1.3. Who uses HLA

Two of the most common use cases for HLA are:

- Training, where humans are trained to perform tasks.
- Analysis, where we can try different scenarios in a simulated world.

Other use cases include test, engineering, and concept development.

HLA was originally developed for defense applications. This is still an important use case for HLA. Examples include training several pilots in flight simulators in motor skills, decision skills, and communications. Another area is command-level training where officers are trained to manage complex situations and command thousands of simulated participants. HLA enables this training to be joint (between Army, Navy, and Air Force) and combined (for example between Navies from different nations).

Training for peace support operations is a related area where defense units can train operations together with police, fire fighters, and non-government organizations such as the Red Cross.

Like many other technologies, originally developed in the defense area, there is a growing civilian user base of HLA. One example is space applications where complex missions, possibly taking months and years, can be simulated in a shorter time and emergency situations can be trained without any real risk. HLA has also been used for space mission control training, such as the docking of automated transfer vehicles with the International Space Station.

Another area is Air Traffic Management where new procedures can be developed, evaluated and trained using a number of interoperable simulations.

Yet other areas include manufacturing, offshore oil production, national railroad

systems, medical simulations, environment, and hydrology.

1.4. HLA is a standard

HLA is an open international standard, developed by the Simulation Interoperability Standards Organization (SISO) and published by IEEE. The development process is open and transparent. Everyone can participate in the development, suggest improvements, take part in the discussions and vote.

HLA is a standards document that describes the components of HLA and what interfaces and properties they must have. Anyone can develop any software component of HLA. Implementations of the different components are today available from commercial companies, governments, academia, and open source developers.

HLA is today a prescribed or recommended standard, for example in NATO as well as by national departments of defense. By establishing such policies, it is possible to facilitate the interoperability of different systems that are acquired over time by an organization.

Another effect of standards is that they enable a marketplace. When systems from different suppliers become interoperable, an end user can select and combine systems that meet his requirements. This reduces the cost, time, and risk for the end user while providing more opportunities for system vendors.

2. The Architecture and Topology of HLA

- The HLA topology is a Service Bus
- Important terms in HLA are: RTI, Federate, Federation, Federation Object Model, and Federation Execution
- The Federation Agreement is a document that specifies the design of a particular federation for a particular purpose
- The Federation Object Model (FOM) is the language of the Federation

2.1. Topologies for Integrating Systems

The previous chapter introduced the concept of a Federation of systems. So how do we connect systems so that they can exchange services in order to meet a common goal?

Figure 2-1 shows two common topologies:

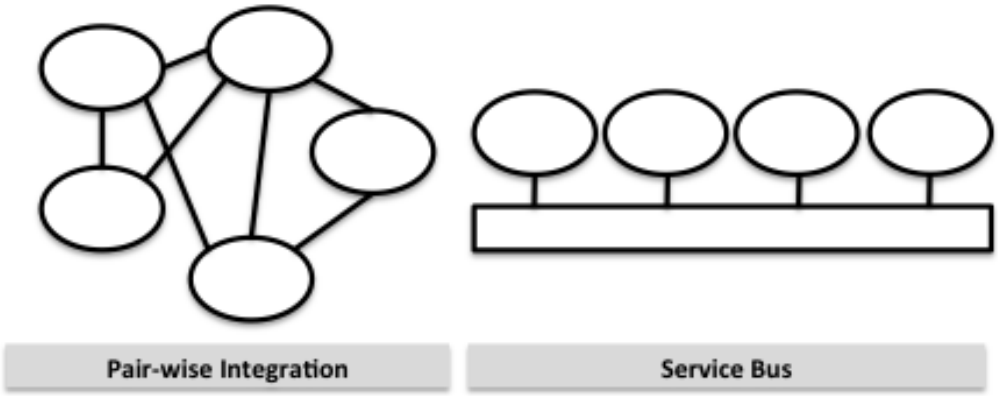


Figure 2-1: Two topologies

When using pair-wise integration you connect systems that need to exchange services in pairs. For each pair of systems, you establish an agreement of what services to exchange. This works well for a few systems, but as the number of systems grows the number of connections will grow exponentially. Another issue is that each system needs explicit knowledge about all other systems that it needs to exchange data with. When you add one new system, you may need to modify a large number of existing systems.

The other type of integration is called Service Bus. Each system has just one connection to the service bus. A common set of services has been agreed upon. Each system provides or consumes the particular services that it is interested in.

Each system can be replaced by another system without updating several other systems. New systems that produce or consume services can easily be introduced. This is more flexible and scalable.

2.2. HLA Terminology

The recommended way to draw a graph for systems that interoperate using HLA is through a “lollipop” diagram, as shown in figure 2-2:

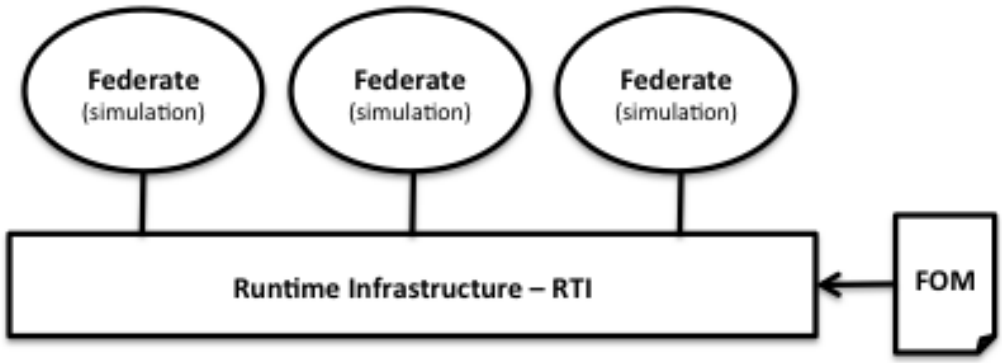


Figure 2-2: The topology of HLA

The basic topology is a number of systems that have one single connection to a service bus that is called the Runtime Infrastructure (RTI). The RTI provides information, synchronization, and coordination services. This type of topology does not force each system to know which other systems that consume or produce information. This approach enables the group of systems to be gradually extended over time and facilitates the reuse of systems in new combinations.

There are five important concepts:

The **Runtime Infrastructure (RTI)**, which is a piece of software, that provides the HLA services. One of them is to send the right data to the right receiver.

The **Federate**, which is a system that connects to the RTI, typically a simulator. Each federate can model any number of objects in a simulation. It can, for example, model one aircraft or hundreds of aircrafts. Other examples of federates are general tools like data loggers or 3D visualizers.

The **Federation**, which is all of the federates together with the RTI that they connect to and the FOM that they use. This is the group of systems that interoperate.

The **Federation Object Model (FOM)** which is a file that contains a description of the data exchange in the federation, for example the objects and interactions that will be exchanged. This can be seen as the language of the federation.

The **Federation Execution**, which is a session when the federation runs, for example a pilot training session when you run several flight simulators. If you run the federation several times you will have several federation executions.

All of these concepts will be used extensively throughout this document and also explained in further detail.

2.3. The Federation Agreement and the Federation Object Model

When you connect several simulation systems you need to decide exactly how the federates are to exchange services. This will be different for example between a flight simulation federation and a medical federation. The exact contract (or design document) for this is called the **Federation Agreement**.

One important part of this is a description of the types of information that needs to be exchanged – the language of the federation. This is described in a Federation Object Model (FOM), which is part of the Federation Agreement.

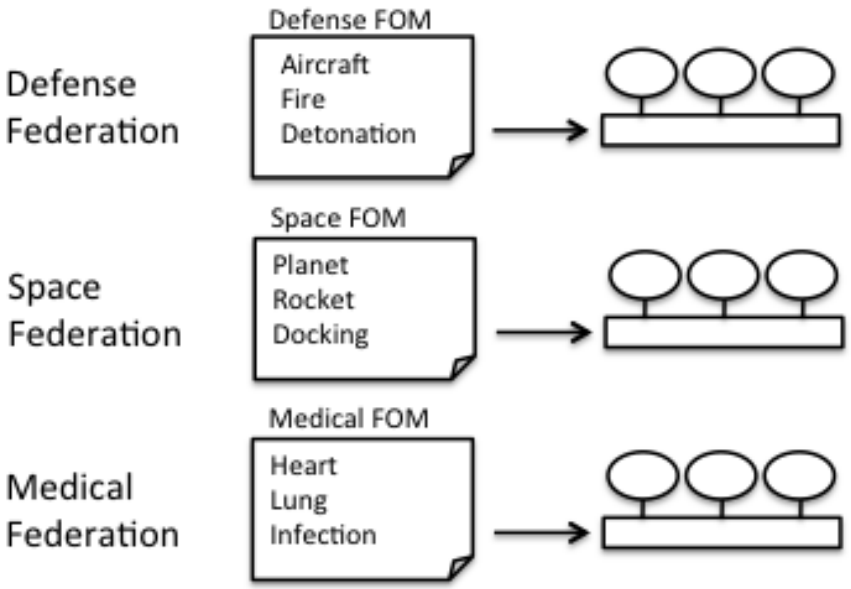


Figure 2-3: Different FOMs for different domains

You need different FOMs when running defense, space, medical, or manufacturing simulations since you need to exchange data about different concepts, as shown in figure 2-3. You can develop a FOM for any domain.

2.4. Content of a FOM

Three of the most important things in the FOM are the Object classes, the Interaction classes, and the Data types, which are shown in figure 2-4.

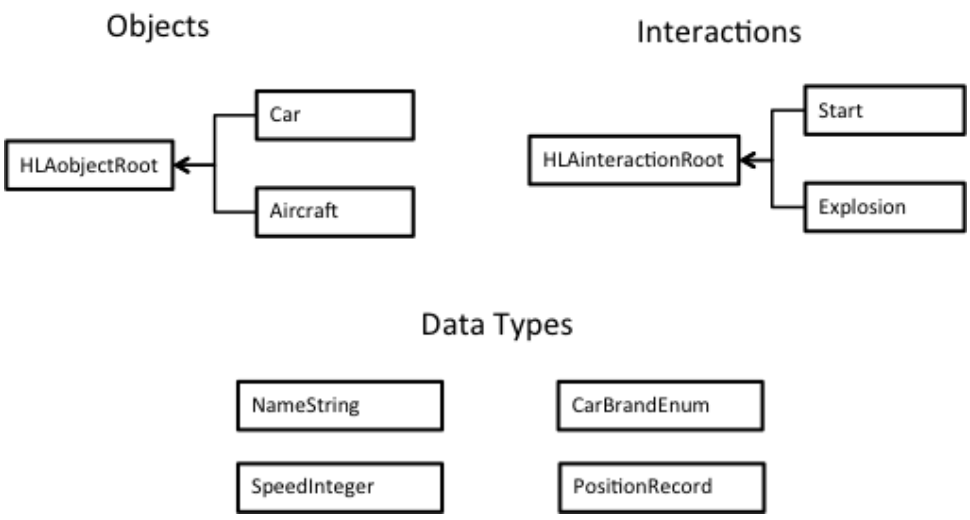


Figure 2-4: Sample contents of an HLA FOM

An **object** is something that persists over time and has attributes that can be updated. A car is a typical object class. Name, position, and speed are typical attributes.

An **interaction** is something that does not persist over time. Start, stop, explosion, and radio messages are typical interaction classes. An interaction usually has some parameters.

The **data types** describe the semantics and technical representation of the attributes of an object class and parameters of an interaction. There are a number of predefined data types in HLA that can be used to build your domain specific data types, including complex data types like records and arrays.

A FOM contains more things, for example general information about the purpose, version and author of the FOM and more.

You can gradually extend a FOM over time without breaking existing simulations. The need to gradually extend the level of interoperability over time is the rule rather than the exception.

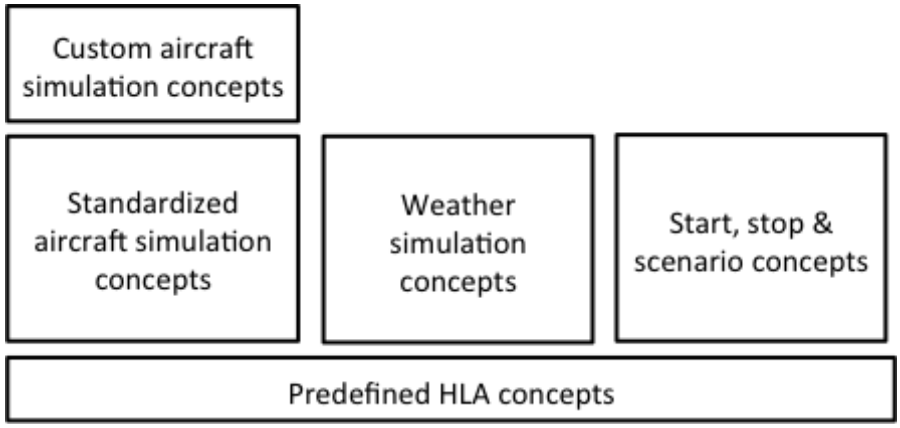


Figure 2-5: FOM Modules

To enable more efficient standardization and reuse, a FOM can be divided into modules. Each module covers a particular concern, for example the information exchange needed between aircraft simulations. It is also possible to extend existing modules with more refined concepts in a new module. In this way a customized aircraft module can build upon a standardized module.

The FOM follows a format called the HLA Object Model Template, which is based on XML. This format is described in the HLA standard. There are also XML Schemas that can be used to verify the format of an object model.

3. HLA Services

- Information services in HLA are based on Publish and Subscribe
- Synchronization services include handling of logical time, synchronization points and save/restore
- Coordination services include keeping track of the federates and the federation, transfer of modeling responsibilities, and advanced inspection and management of the federation.
- The formal standard consists of three documents: Rules, Interface Specification, and Object Model Template.

3.1. HLA Services

HLA describes a number of services that are implemented by the RTI. In HLA they are divided into seven service groups in the standard. In this introduction we categorize them further into

- Information exchange services
- Synchronization services
- Coordination services

To use these services a federate will make calls to the RTI and receive callbacks from the RTI, which means that there is a two-way communication.

3.2. Information Services

These services enable federates to exchange data according to the FOM using a Publish/Subscribe scheme.

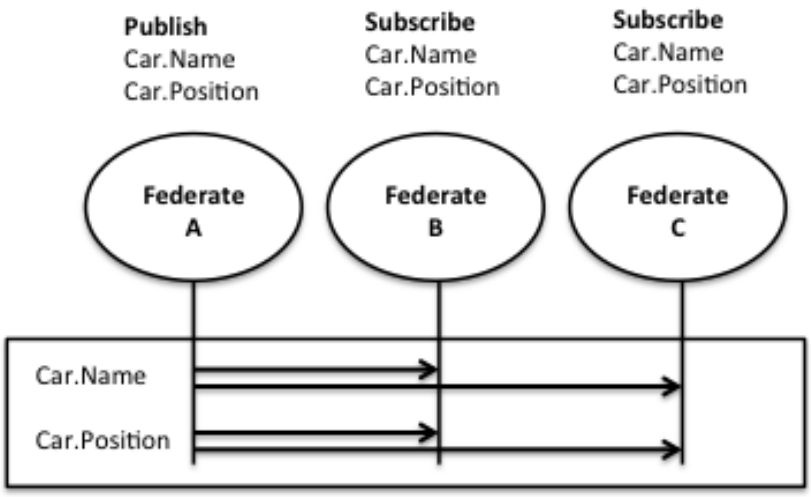


Figure 3-1: Publish and Subscribe

The RTI keeps track of which federates that subscribe to each type of object, attribute, or interaction, which means that the federates want to receive that type of data. It also keeps track of which federates that publish each type of object, attribute, or interaction, which means that they are able to send that type of data. Finally the RTI acts like a switchboard and delivers data from publishers to subscribers as shown in figure 3-1.

To illustrate this you can look at this "Publish/Subscribe Matrix" with a car simulator, a data logger and a map viewer. Federates are listed in the top row and attributes of the car object are listed in the left column.

Attribute	**CarSim**	**Data Logger**	**Map Viewer**
Car.Name	Publish	Subscribe	Subscribe
Car.Position	Publish	Subscribe	Subscribe

Figure 3-2: Publish/Subscribe Matrix

The simulator CarSim publishes the name and position of cars. The data logger and the map viewer subscribe to the name and position of cars. The RTI will make sure that any name and position of a car will get delivered to the data logger and map viewer. It is not necessary for the CarSim to keep track of which federate that wants to get car information at any given time. The RTI handles this automatically.

It is possible to exchange hundreds of thousands of updates per second between standard PCs using a modern RTI. Typical latency is in the range of a millisecond.

There is also a more advanced type of filtering based on the data values rather than the class of the data, called Data Distribution Management. This can be used for example when a federate only wants to get updates for cars in a particular geographical area. This enables federations to run large scenarios while limiting the load on each simulator.

3.3. Synchronization Services

There are three types of synchronization services:

- Handling of logical time. Some federations run in real time. Others may run faster or slower than real time, possibly simulating several weeks of a scenario in just a few minutes. To be able to correctly exchange simulation data with time stamps, the RTI can coordinate both how fast the simulators advance logical or scenario time as well as correct delivery of time stamped data.

- Synchronization points that enables the members of the federation to coordinate when they have reached a certain state, for example when they are ready to start simulating the next phase of a scenario.

- Save/restore that makes it possible to save a snapshot of a simulation at a given time. This is useful when you want to go back to a certain point in time to rerun a scenario with different parameters. It is also useful for backup purposes.

3.4. Coordination Services

These services include:

- Management of federation executions and federates that are joined to a federation executions.
- Transfer of modeling responsibilities. This means that one simulator, that simulates aircrafts, can hand over the responsibility for updating a particular aircraft to another simulator at runtime.
- Advanced inspection and management of the federation. These services are provided in a separate set of services described in the Management Object Model.

3.5. Service grouping in the HLA standard

The HLA standard lists the services in a slightly different order from the above. It also contains a number of utilities called Support Services. Here is the full list of services again, in the order that the HLA standard describes them:

Federation Management	Keep track of federation executions and federates, synchronization points, save/restore
Declaration Management	Publish and subscribe of object and interaction classes
Object Management	Registering and discovering object instances, updating and reflecting attributes, sending and receiving interactions
Ownership Management	Transfer of modeling responsibilities
Time Management	Handling of logical time including delivery of time stamped data and advancing federate time
Data Distribution Management	Filtering based on data values
Support Services	Utility functions
Management Object Model	Inspection and management of the federation

3.6. About the Standards documents

The HLA standard consists of three parts:

1. The “Rules” (IEEE 1516-2010) that contains ten rules for the federates and the federation
2. The “Interface Specification” (IEEE 1516.1-2010) that describes the RTI services in detail
3. The “Object Model Template” (IEEE 1516.2-2010) that describes the format for FOMs.

The purpose of the standard is to give a full and exact specification of the High-Level Architecture whereas this document explains the standard and how to use it without providing every detail.

You should consider getting a copy of these standards. In particular the Interface Specification is strongly recommended when developing federates. The recommended way to get these standards is to become a SISO member since this gives you access to all IEEE standards developed by SISO. You may also visit the IEEE web site and buy them.

4. Connecting to a Federation

- The Fuel Economy Federation is introduced. The Federation Agreement is available in Appendix A.
- HLA functionality of an application should be separated out into an HLA module
- Initially your federate needs to Connect, Create a Federation Execution and Join.
- Finally the federate needs to Resign, Destroy the Federation Execution and Disconnect

4.1. Getting Connected

This chapter shows how to connect to the federation and how to leave it. It gives an overview of the services without getting into detail about all exceptions or data types. For a complete description, see the HLA standard. You may also want to look at the C++ and Java APIs and code samples at the end of this chapter.

4.2. An Overview of the Example Application

The Fuel Economy Federation is used to evaluate how far cars from different manufacturers can drive using a limited amount of fuel. Each car manufacturer provides a simulator that simulates selected models. Here is a diagram of the federation.

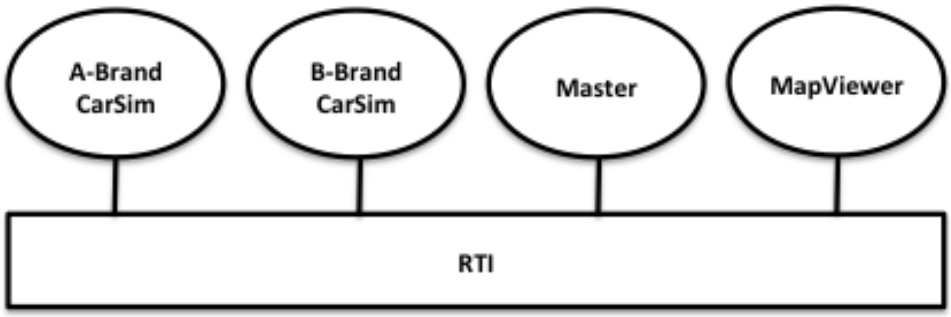


Figure 4-1: The Fuel Economy Federation

In this case A-Brand will simulate their 319 and 440d models whereas B-Brand will simulate their 4-8 and MountainCruiser models. There will be more car simulators later so the design will not be locked to these specific cars. There will be one management federate called Master from which an operator can set up the scenario and start and stop the simulation. There will also be a federate called MapViewer that displays a map with the cars and their fuel status.

The following information is available about our example federation:

- Appendix A: The Federation Agreement
- Appendix B: The Federation Object Model (FOM)
- Appendix C: A description of the Federates and the File Formats.

You are encouraged to refer to them while reading this tutorial.

4.3. Practical Exercise

A lab that lets you test this federation on your own computer is provided in Appendix D-1

4.4. An HLA Module for Your Simulation

You usually add HLA functionality to a new or existing simulation in a separate module. We have chosen to call it the HLA module.

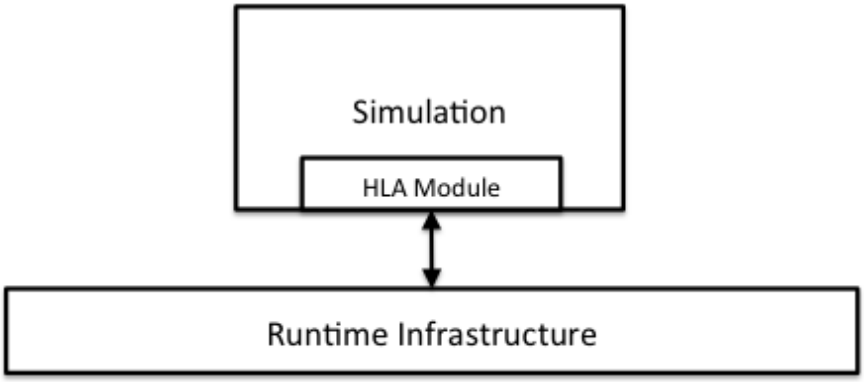


Figure 4-2: The HLA module of a simulation

It may be tempting to develop one single HLA module that fits all of your simulations. Such an HLA module would subscribe to all kinds of data in the FOM. This works for smaller federation but will unfortunately limit the scalability of your federation. Best practice is to develop an HLA module that focuses on only the data that one particular federate (or class of federate) needs to publish and subscribe.

This tutorial describes how to develop the HLA module. The HLA module performs calls according to the HLA standard to the Runtime Infrastructure (RTI). You will also need to write code that connects the internals of your simulation with the HLA module. Note that for some simulations you may choose to make the HLA module optional, which means that your simulation can run both with and without an HLA connection.

4.5. The Central and Local RTI components

An RTI consists of two types of software components, as shown in the following picture:

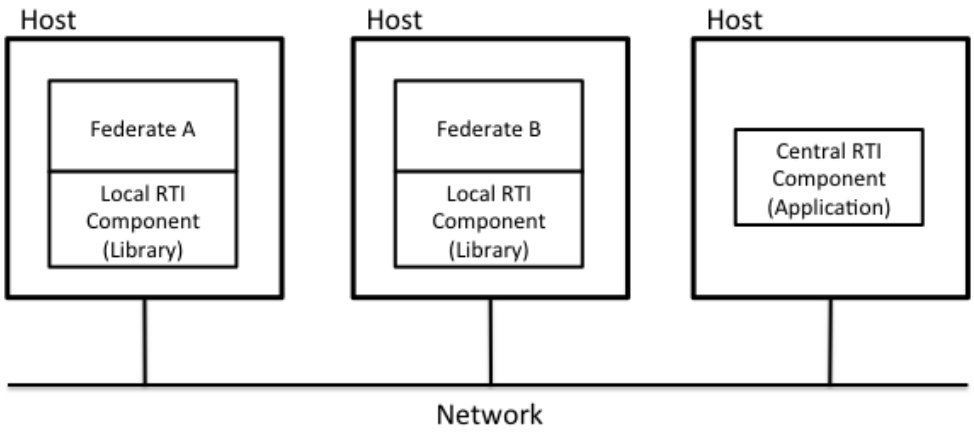


Figure 4-3: The CRC and the LRC

The **Local RTI Component** (LRC), is a local library installed on each computer with a federate. For C++ federates this is a “dll” or ”so” file. For Java federates this is a “jar” file.

The **Central RTI Component** (CRC), is an application that coordinates the federation execution and keeps track of the joined federates. The CRC is a program that needs to be started at a computer with a well-known address. You need to start the CRC before any federate can join a federation. The CRC user interface is a good point to get an overview of the federation.

You may have several federates as well as the CRC on the same computer if you wish. Note that you must install the LRC libraries on each computer with a federate.

4.6. Connecting and Joining

When a simulation is part of a federation it is called a federate. The first thing that your federate needs to do is to call the RTI (actually the LRC) in order to connect to the RTI.

It then needs to join a Federation Execution, which always has a unique name, for example “MyAirSimulation” or “TrainingSession44”. There may exist many Federation Executions at the same time but a typical simulation will only join one Federation Execution. It may be necessary to create a Federation Execution if it doesn’t exist. These are the services we need to use to create the federation execution and to join it:

```
rti = RTIambassadorFactory.createRTIambassador()
rti.connect(federateAmbassador, IMMEDIATE, “MySettingsDesignator”)
rti.createFederationExecution(“MyFederation”, “MyFOM”)
rti.joinFederationExecution(“MyName”, “MyFederateType”, “MyFederation”)
```

4.7. A look at the RTI

By looking at the user interface of the RTI, it is easy to verify that the federation execution has been created and that the federate has joined.

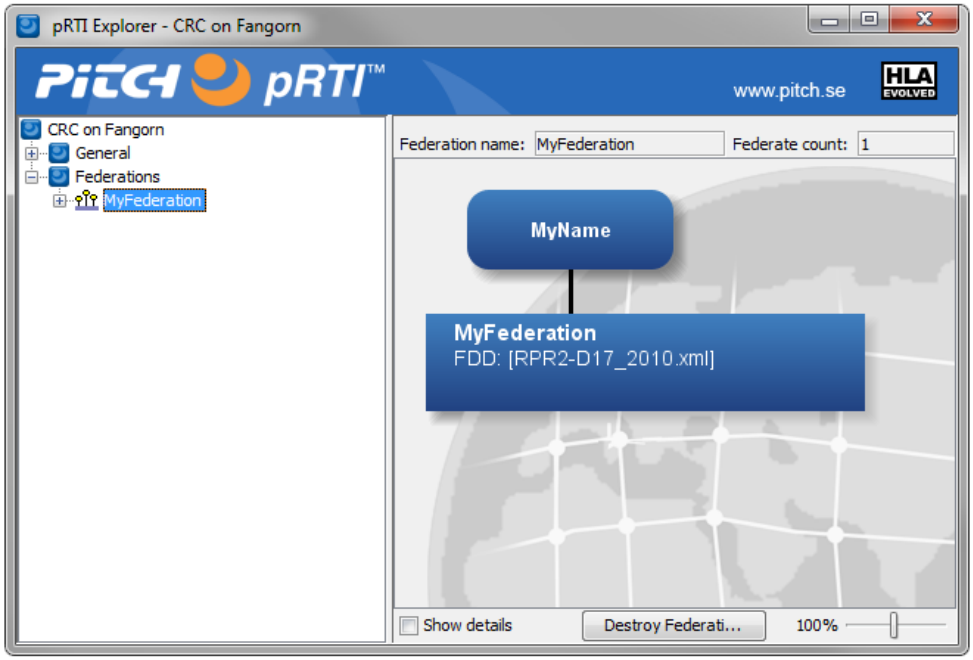


Figure 4-3: The federate and the federation in the RTI GUI

4.8. Resigning and Disconnecting

At the end of the program we need to resign from the federate, destroy the federation execution, and finally disconnect. We will then use the following services:

```
rti.resignFederationExecution(CANCEL_THEN_DELETE_THEN_DIVEST)
rti.destroyFederationExecution()
rti.disconnect()
```

Actual code is available in the samples directory of the HLA Evolved Starter Kit.

4.9. Calls and Callbacks

There will be both calls to the RTI and callbacks from the RTI to your federate. You will use an object called the RTI ambassador for making calls to the RTI. It is created using an RTIambassadorFactory in section 4.6. You need to supply a Federate Ambassador, as shown in the Connect call above, that the RTI will make callbacks to.

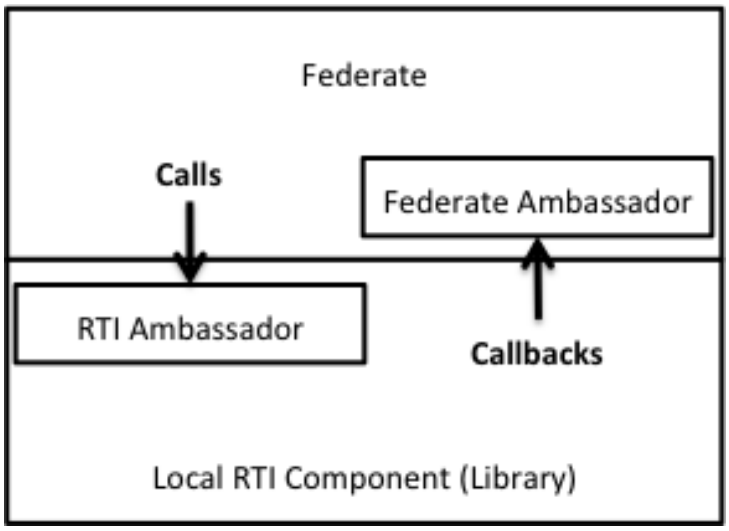


Figure 4-4: Calls, callbacks and ambassadors

There are two callback models: Immediate, which means that the RTI will deliver the callbacks in a separate thread as soon as they are received. The other model is called Evoked which means that you explicitly call the Evoke method to get callbacks delivered. We suggest using the Immediate mode. Less advanced developers may choose to use the Evoked model since it uses only one thread for calls and callbacks.

We will now discuss the above services in more detail.

4.10. HLA Service: Connect

This service connects your simulation to an RTI. The parameters for this connection, like network address, are provided either through a parameter (a string) called the Local Settings Designator, a configuration file, environment variables, or a combination of these. For best flexibility, avoid hard coding this parameter into your program. You also need to specify whether callbacks from the RTI should be performed immediately or if they should be explicitly evoked. Be careful to handle the Connection Failed exception, which typically occurs when there is no RTI available given the specified network address. If the Connect service throws an exception there is no use trying any other HLA service before you have successfully connected.

Read more about Connect in section 4.2 of the Interface Specification.

4.11. HLA Service: Create Federation Execution

This service creates a federation execution with a specific name. You may try to create a federation execution every time since there are no major problems with trying to create a federation execution that already exists. The only result will be an exception saying that it already exists, which in many cases can be safely ignored. So be sure to catch the “The specified Federation Execution already exists” exception and handle it silently. You need to provide a valid FOM as well,

actually a list of one or more FOM modules. Be sure to watch for exceptions for not being able to locate the FOM module as well as for invalid FOM modules. There are also some additional parameters, for example time representations that will be described in a later chapter.

Read more about Create Federation Execution in section 4.5 of the Interface Specification.

4.12. HLA Service: Join Federation Execution

This service makes your simulation a member of a federation execution. You need to provide the name of the Federation Execution that you want to join. You should also provide the name and type of your federate. Be sure to handle the exceptions when the federate name is already in use by another federate. There are additional parameters that will be described later.

Read more about Join Federate Execution in section 4.9 of the Interface Specification.

4.13. HLA Service: Resign Federation Execution

This service resigns the federate from the federation, i.e. your simulation will no longer be a member of the federation execution. It requires a directive that specifies what should happen with, for example, objects that the federate has created but not deleted. Unless you have special requirements it is recommended that you use the CANCEL_THEN_DELETE_THEN_DIVEST directive.

Read more about Resign Federation Execution in section 4.10 of the Interface Specification.

4.14. HLA Service: Destroy Federation Execution

This service destroys the federation execution. You may try to destroy the federation execution every time you have resigned. If there are still other federates in the federation this operation will fail and the “Federates are joined” exception will be thrown, which in many cases can be safely ignored.

Read more about Destroy Federation Execution in section 4.6 of the Interface Specification.

4.15. HLA Service: Disconnect

This service disconnects your federate from the RTI. It is the last service you call to the RTI. After this call you cannot make any RTI calls until you have performed a connect call.

Read more about Disconnect in section 4.3 of the Interface Specification.

4.16. Additional comments

Once your federate is connected it can also call the List Federation Executions

service, which returns a list of all available federations for this RTI.

Your federate may also become disconnected during the execution due to external issues like communication failures, broken cables, etc. In this case all RTI calls will throw the Not Connected exception. In this case you need to Connect again. This is described in the Fault Tolerance section in a later chapter.

4.17. Practical Exercise

A lab for this chapter is provided in Appendix D-2

5. Interactions – Creating the FOM

- A typical FOM contains an identification table, object classes, Interaction classes, and data types.
- The identification table describes things like the purpose, author and, version of a FOM
- Interactions with parameters are described in a class tree
- A predefined FOM module called the MIM contains standard data types like HLAunicodeString
- For integers and floats we need to create Simple Data Types

5.1. Overview

We will now create a FOM for this federation. It will describe the information that needs to be exchanged between federates. The FOM will thus not contain all of the internal data of the federates. The exchanged data will be modeled as object classes and interaction classes. Object classes have attributes that can be updated over time. Interaction classes only exist for a short moment. Note that the FOM describes object classes, not the object instances.

The following shared information will be modeled in our sample:

- A number of messages to start and stop the simulation and to manage the scenario will be modeled as Interaction Classes.
- A number of data types for attributes and parameters will also be modeled
- In a later chapter we will model Cars using Object Classes.

5.2. A First Look at the FOM

Before we can start exchanging data we need a FOM, which this chapter takes a closer look at. The FOM is implemented as an XML file but it can also be displayed as tables, for example in the HLA standard or in various report formats. In this case we will visualize the FOM in the FOM editing program Pitch Visual OMT.

We will now develop the following items:

- The Identification table
- Interactions with parameters
- Data types

5.3. The Identification table

First of all we need to provide the name, purpose, version and author of the FOM.

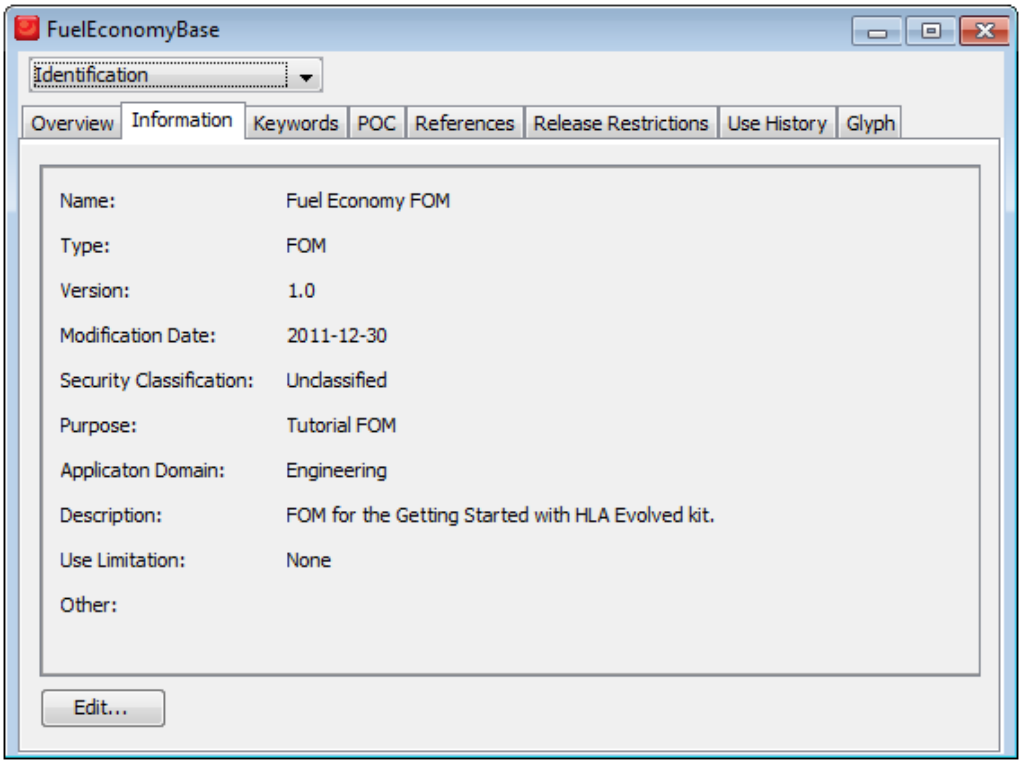


Figure 5-1: Identification table

It is strongly recommended that you fill in at least this part of the identification with name, purpose, date and version, and that you provide information about yourself in the Point of Contact section. You should also consider applying configuration control of your FOM for example in a version tracking system.

Read more about the Identification Table in section 4.1 of the Object Model Template Specification.

5.4. Interactions with parameters

We will start by defining five Interaction Classes. They are all subclasses of the predefined class HLAinteractionRoot.

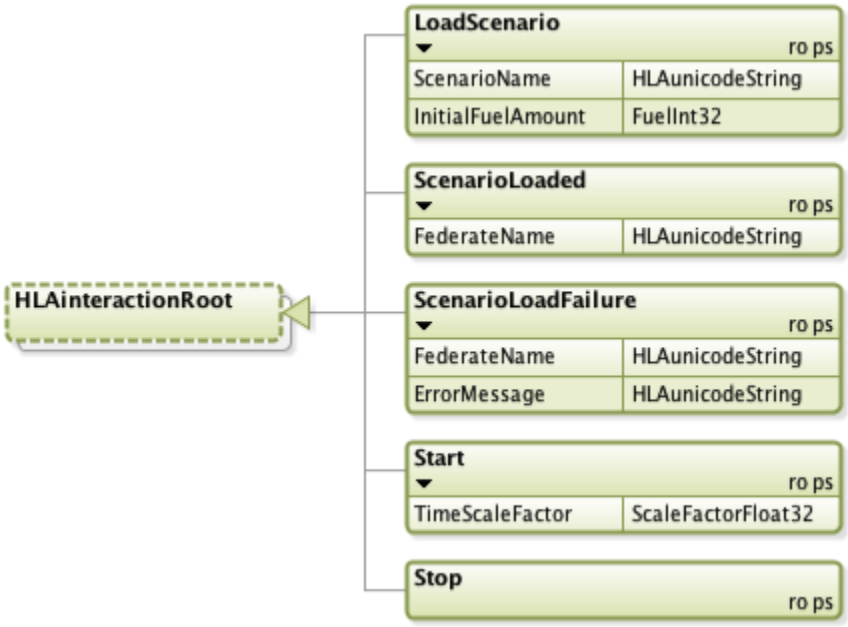


Figure 5-2: Interaction classes

The LoadScenario is used to inform all participating federates about the scenario to run. It provides the name of the destination as well as the amount of fuel to be filled before starting. The ScenarioLoaded interaction is used by the federates to confirm that the scenario has been loaded. There is also an interaction called ScenarioLoadFailure to indicate that the scenario could not be loaded. There are also start and stop interaction to control the execution.

Let's take a closer look at the LoadScenario interaction.

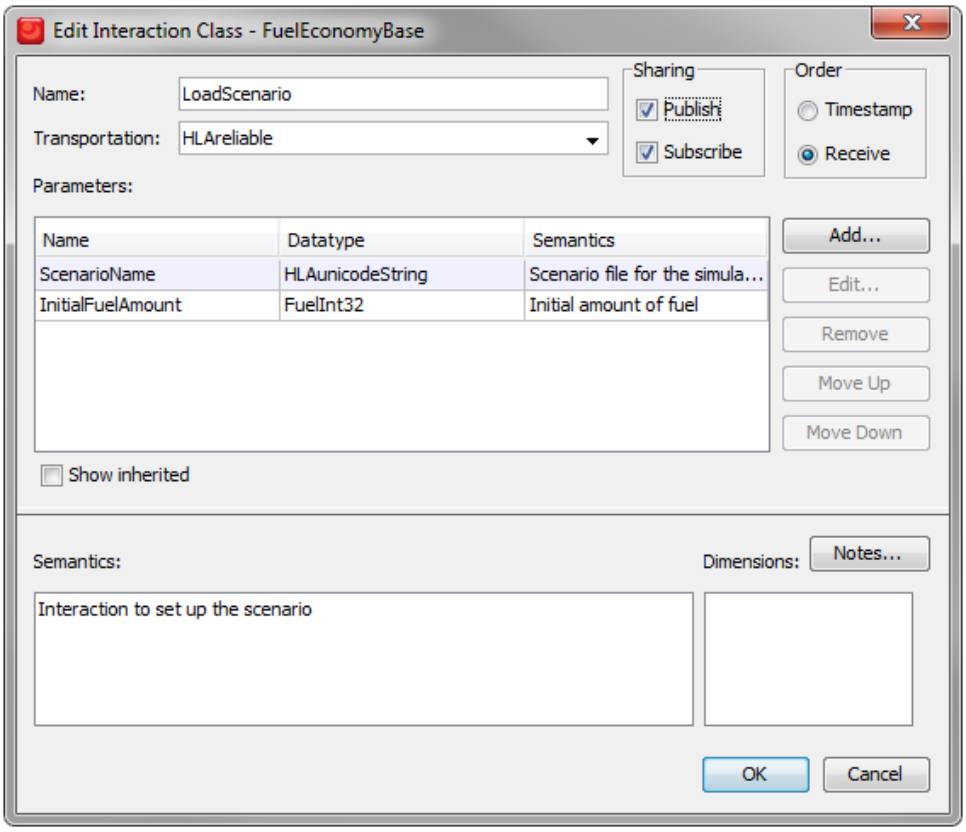


Figure 5-3: The LoadScenario interaction

This interaction will be both Published and Subscribed to by federates in the federation. There are two parameters:

- **ScenarioName**, which uses the predefined data type HLAunicodeString.
- **InitialFuelAmount,** which uses a user-defined data type called FuelInt32.

There are also additional properties of an interaction that will be introduced later in this document.

Read more about Interactions and Parameters in section 4.3 and 4.5 of the Object Model Template Specification.

5.5. A simple data type

We will need to define a data type for the fuel levels. By having one common data type for fuel we can ensure a clear and consistent definition for all parameters and attributes that relate to fuel levels. For simple data, which consists of just one integer or float, we use the Simple data type. This is what the FuelInt32 data type looks like:

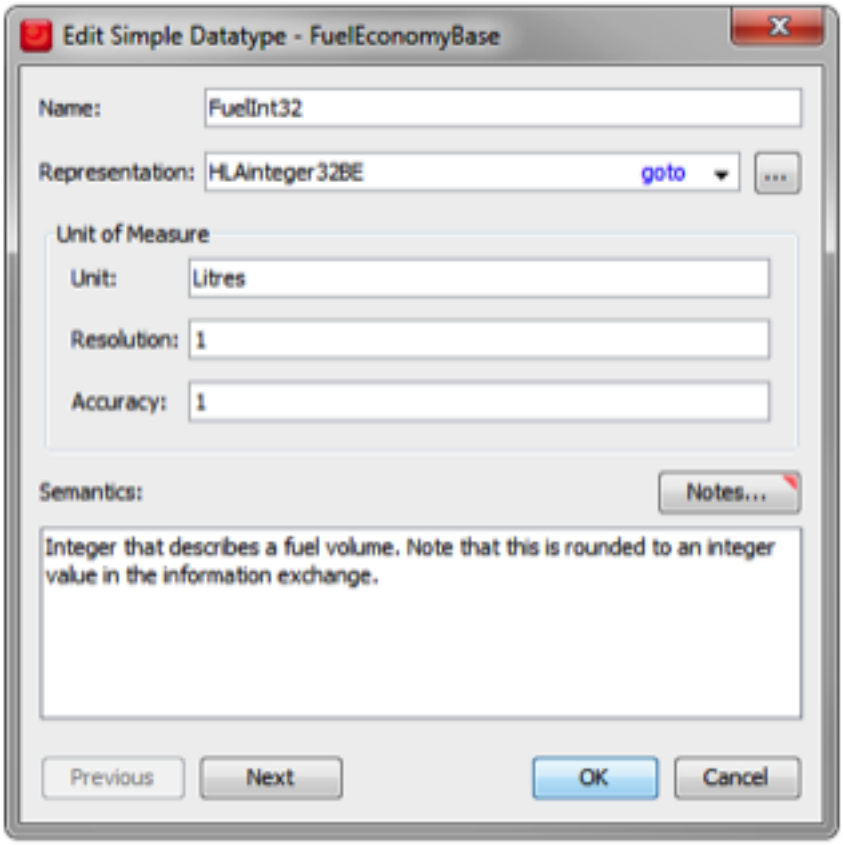


Figure 5-4: The FuelInt32 Simple data type

Note the following fields:

- The **Representation** describes the exact representation of the data when it is exchanged between different federates. In this case the predefined representation HLAinteger32BE is used, which is a 32 bit integer with big-endian representation. In this federation we have chosen to use mainly 32 bit integers and floats with big-endian representation.
- The **Unit** for measuring fuel is the metrical liter.
- The **Resolution** of the value is one liter.
- The **Accuracy** (i.e. correctness as compared to the real value) is one liter.

This dialog may at first seem to be unimportant and overly detailed. It is actually extremely important. Sending 16 bits of data to a federate that expects 32 bits may crash that federate. Sending fuel levels in gallons instead of liters will make the output of the federation useless. Sending data with too low resolution or accuracy may cause incorrect computations.

Read more about Simple data types in section 4.13.4 of the Object Model Template Specification.

5.6. More about FOMs

The FOM can be provided as one single file or, for larger FOMs, as several FOM

modules. A FOM module is simply a number of related object classes, data types, etc, that are stored in a separate file. It is not uncommon to have FOM modules that extend existing FOMs by providing subclasses. Modular FOMs simplify both the development and maintenance of FOMs as well as their reuse.

There is one special module called the Management and Initialization Module (MIM). It contains all of the predefined concepts of HLA. We have already seen the HLA Object Root, the HLA Interaction Root, as well as HLAunicodeString and HLAinteger32BE. The MIM is provided automatically by the RTI when a federate calls the Create Federation Execution service.

As you may have noticed, the classes and data types in the MIM all have names that start with “HLA”. Only MIM concepts are allowed to start with “HLA”.

We have also used a naming convention for data types where they start with a description of what it measures (Fuel, Angle, Position) and end with the technical representation, for example FuelInt32, AngleFloat64, FuelTypeEnum32, and PositionRec. Note also that we have avoided using the unit or resolution in the name. This type of convention is convenient but not required by the standard.

5.7. Practical Exercise

A lab for this chapter is provided in Appendix D-3

6. Interactions – Calls for Sending and Receiving

- To be able to reference an Interaction class we need to retrieve a handle for the class
- To be able to send and receive interaction of a particular class we need to publish and subscribe to that class
- Before the data can be sent we need to encode it according to the FOM, for example using the Encoding Helpers

This chapter tells you how to send and receive interactions from a federate. We will use the LoadScenario interactions as an example since it contains two parameters. To handle interaction, we will cover the following steps:

- Some initial preparations that need to be done.
- How to send an interaction.
- How to receive an interaction.

6.1. Initial preparations for Interactions

Initially we will need to do three things before we start the main simulation loop:

1. Allocate helper objects for correctly encoding and decoding data according to the FOM.
2. Get “handles”, which is a type of reference, for the Interaction Class and its Parameters.
3. Publish and subscribe to the interaction class.

Here is the pseudocode:

```
HLAunicodeString myStringEncoder
HLAinteger32BE myInt32BEencoder

ParameterHandleValueMap parameters
VariableLengthData userSuppliedTag

scenarioInteractionHandle = rti.getInteractionClassHandle
                                    (“LoadScenario”)
destinationParameterHandle = rti.getParameterHandle
                                    (scenarioInteractionHandle, “Destination”)
initialFuelAmountParameterHandle = rti.getParameterHandle
                                    (scenarioInteractionHandle, “InitialFuelAmount”)

rti.publishInteractionClass(scenarioInteractionHandle)
rti.subscribeInteractionClass(scenarioInteractionHandle)
```

First we create some Encoding Helper objects. These helper classes in HLA makes it easy to encode and decode data correctly. For performance reasons we don't want to create them each time we use them, so we do this before we get into the main loop. We will also need a ParameterHandleValueMap when we send the interaction as well as a userSuppliedTag. We will also create an empty userSuppliedTag since this concept is not used in this example.

We then need to fetch handles for the LoadScenario interaction in the FOM as well as the Destination and InitialFuelAmount parameters. These handles are used in the RTI calls. Note that they use the concepts from the FOM we created in the previous chapter. Finally we publish and subscribe to this interaction. In this federation, it will be the Master that publishes this interaction class and the CarSims that subscribe to it, but this example shows both calls. The publish and subscribe calls are very important in HLA since they tell the RTI about which federates that will send certain types of information and to which federates they should be delivered.

Here is some more information about these services.

6.2. Encoding and Decoding Helpers

These are formally not seen as RTI services. They are utilities that are described in the Programming Language Mappings and the C++ and Java APIs.

Read more about Encoders and Decoders in section 12.11.3.2 and 12.12.4.2 of the Interface Specification.

6.3. HLA Service: Get Interaction Class Handle

This service returns a handle for the specified interaction class in the FOM. You are allowed to omit the initial "HLAinteractionRoot" but otherwise the class name shall be fully specified. Note that this service (and many other Get Handle services) may throw a "not defined" exception meaning that there is nothing in the FOM that matches this string.

Read more about Get Interaction Class Handle in section 10.15 of the Interface Specification.

6.4. HLA Service: Get Parameter Handle

This service returns a handle for the specified parameter of an interaction. The interaction class handle needs to be supplied.

Read more about Get Parameter Handle in section 10.17 of the Interface Specification.

6.5. HLA Service: Publish Interaction Class

This service informs the RTI that the federate publishes the specified interaction,

which means that it can send such interactions.

Read more about Publish Interaction Class in section 5.4 of the Interface Specification.

6.6. HLA Service: Subscribe Interaction Class

This service informs the RTI that the federate subscribes to the specified interaction, which means that the federate will be notified whenever another federate sends such an interaction.

Read more about Subscribe Interaction Class in section 5.8 of the Interface Specification.

6.7. A look at the RTI

By looking at the user interface of the RTI it is easy to verify that the federate has indeed published the interaction.

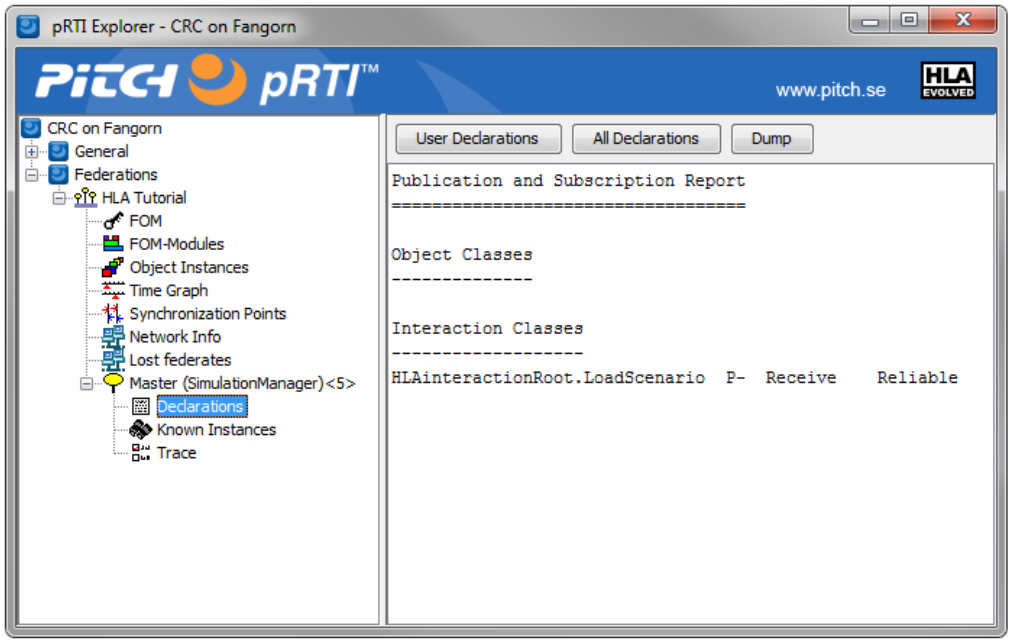


Figure 6-1: An interaction class has been published as seen in the RTI GUI

6.8. Code for Sending Interactions

To send an interaction, we will encode the parameter values and send the interaction. In this example the destination will be San Jose and the initial fuel amount will be 40 liters. Here is the pseudo code:

```
myStringEncoder.setValue("San Jose")
myInt32BEencoder.setValue(40)

parameters[destinationParameterHandle] = myStringEncoder.encode()
parameters[initialFuelAmountParameterHandle] = myInt32BEencoder.encode()

rti.sendInteraction(scenarioInteractionHandle, parameters, userSuppliedTag)
```

Note that we use the encoders that we created earlier and assign the desired values, i.e. San Jose and 40. This is done with the "=" operator in C++ and "setValue" in Java. We then build the parameter structure which is a map consisting of pairs of parameter handles and encoded values. By calling the encode method for an encoder we will get a correctly encoded byte array, no matter how complex the data to be encoded is. It may be slightly overkill to use encoding helpers for an integer but this will guarantee that the encoded result is correct.

When we send the interaction we can also provide a user supplied tag. In this case it is empty.

6.9. HLA Service: Send Interaction

This service sends an interaction and the supplied parameter values. A user supplied tag can be provided.

Read more about Send Interaction in section 6.12 of the Interface Specification.

6.10. Receiving Interactions

The RTI will deliver interactions to our federate by making calls to our Federate Ambassador, also known as a callback. This is the first time that we show a callback in this tutorial. There is a predefined Federate Ambassador called the NullFederateAmbassador with no functionality. It is recommended that you subclass the NullFederateAmbassador and then override selected methods. In this way your federate will support all callbacks even if you haven't provided any code for all callbacks.

To handle an incoming interaction we need to decide which type of interaction this is and then decode its parameters. We recommend using separate encoding helper objects for calls and callbacks. Here is some pseudocode:

```
Method FederateAmbassador.ReceiveInteraction(theInteraction, theParameterValues,
                                                    TheUserSuppliedTag)

    IF theInteraction=scenarioInteractionHandle THEN
      myStringEncoder.decode(theParameterValues[destinationParameterHandle])
      wstring ws = myStringEncoder
      myInt32BEencoder.decode(theParameterValues [initialFuelAmountParameterHandle])
       int fuel = myInt32BEencoder.getValue()
    END IF
```

First we check which type of interaction this is. If it is the LoadScenario interaction then we can fetch the parameter values. We pass them to the decode method of the encoders. This code assumes that we always get both parameters. A safer approach is to loop through the parameters to check which parameters that are present. Note that the decoders may also throw exceptions if the incoming data is incorrect.

6.11. HLA Service: Receive Interaction (callback)

In this example we have simplified the parameter list. There are also optional parameters like a time stamp. Note also that there are actually three different versions of the ReceiveInteraction method for the FederateAmbassador in the APIs. Different versions will be called depending on how many optional parameters that are present. It is highly recommended that you dispatch the calls that you don't use, in this case the ones with additional optional parameters, to the version of the service that you have actually implemented.

Read more about Receive Interaction in section 6.13 of the Interface Specification.

6.12. Additional Comments

Sending interactions is quite easy but receiving interactions is much more difficult for several reasons:

- Your code can decide how many interactions that your federate sends but your federate has no control over how many interactions that it will receive in a federation. Received interactions will impose additional workload on your federate.
- You don't know exactly when they arrive so you will need to decide when and how they are handled.
- You don't know how correct the information in incoming interactions is. Experienced federate developers always take precautions and handle decoding exceptions as well as checking the correctness of decoded values.

6.13. Practical Exercise

A lab for this chapter is provided in Appendix D-4

7. Objects – Creating the FOM

- Objects with attributes are described in a class tree
- For enumerated values like the FuelType we use an Enumerated Data Type
- For complex data types, like a position with Lat and Long we use an HLAfixedRecord data type.
- For integers and floats we need to create Simple Data Types

7.1. The Car Object Class

We will now define the Car Object Class. It contains a number of attributes for the car that will be updated over time. This is what it looks like:

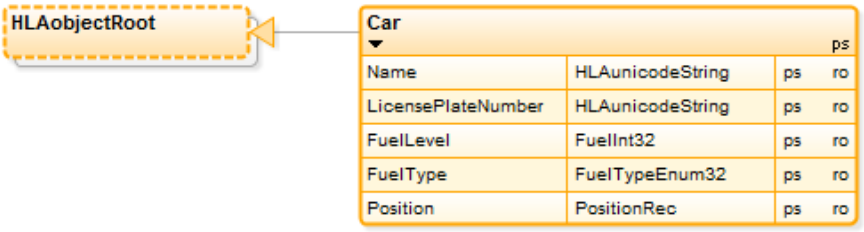


Figure 7-1: Object Classes

The Car object class is a subclass of the predefined HLAobjectRoot. Let's take a closer look at it:

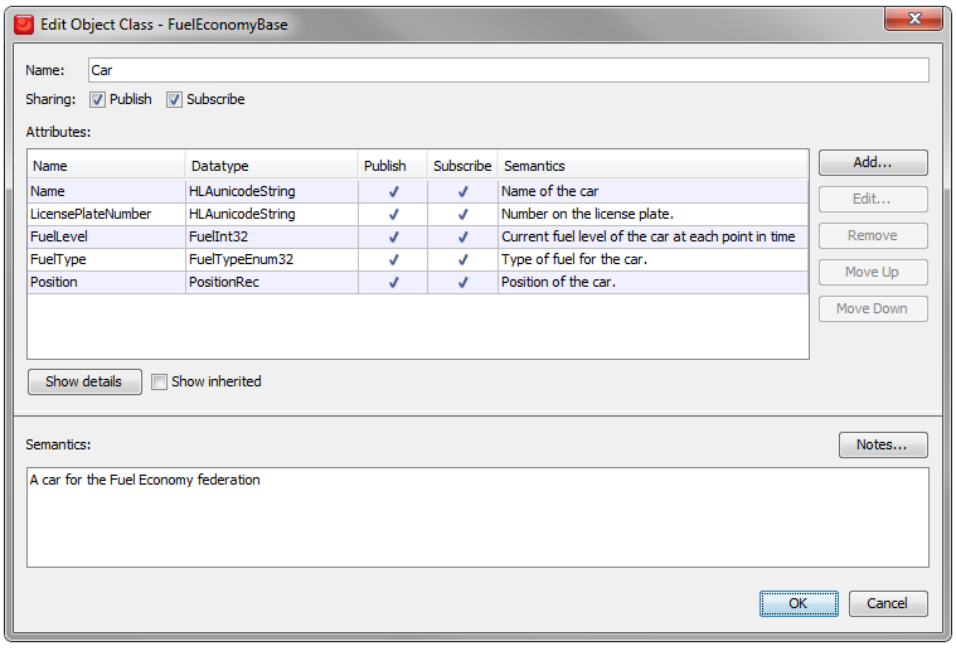


Figure 7-2: The Car Object Class

This class will be both Published and Subscribed by federates in the federation. There are five attributes:

- **Name** and **LicensePlateNumber** which use the predefined data type HLAunicodeString.
- **FuelLevel** which use the FuelInt32 that is also used in one of the interactions.
- **FuelType** which describes type of fuel used by the car. This is an enumeration that we will soon look at in detail.
- **Position** which is a Record that we will also look at in detail.

There are also additional properties of a class that will be introduced later in this document.

Read more about Object Classes and Attributes in section 4.2 and 4.4 of the Object Model Template Specification.

7.2. A closer look at the Name attribute

Let's take a loser look at the Name attribute of an object class:

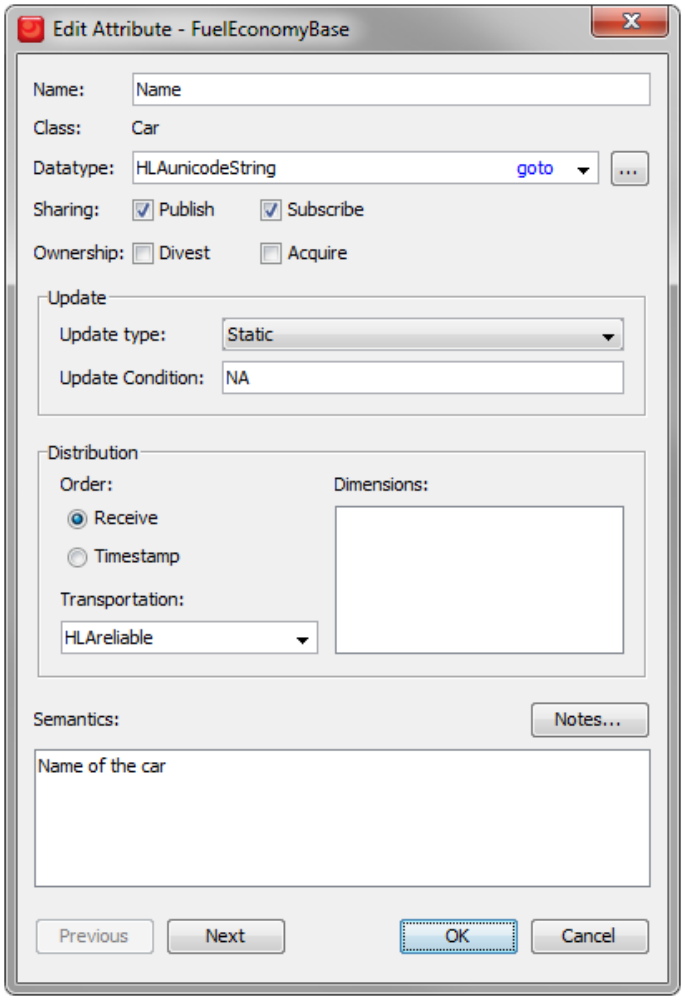


Figure 7-3: The Name Attribute

In this detailed view you can also see the update type, which for the is Static. Other update types are Conditional and Periodic.

There are also additional properties of an attribute that will be introduced later in this document.

Read more about Attributes in section 4.4 of the Object Model Template Specification.

7.3. A closer look at the FuelType attribute

The Fuel Type attribute looks like this:

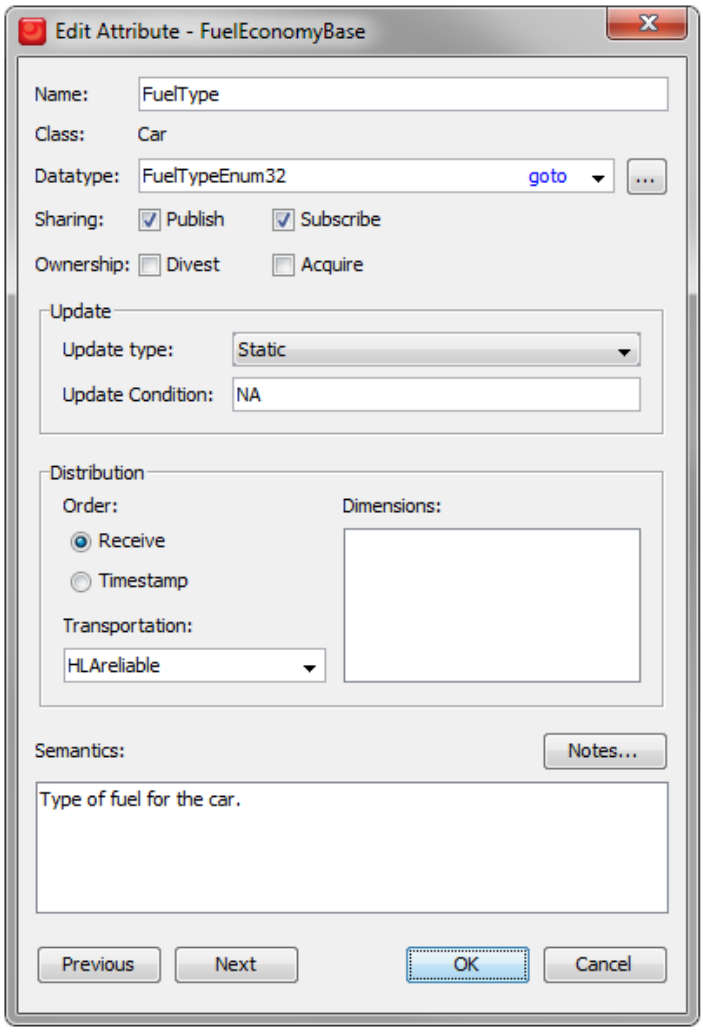


Figure 7-4: The Fuel Type attribute

The Update Type is Static, just like the Name and LicensePlateNumber. As you can see the FuelType is an enumerated value usint the data type FuelTypeEnum32. Let's take a closer look at it:

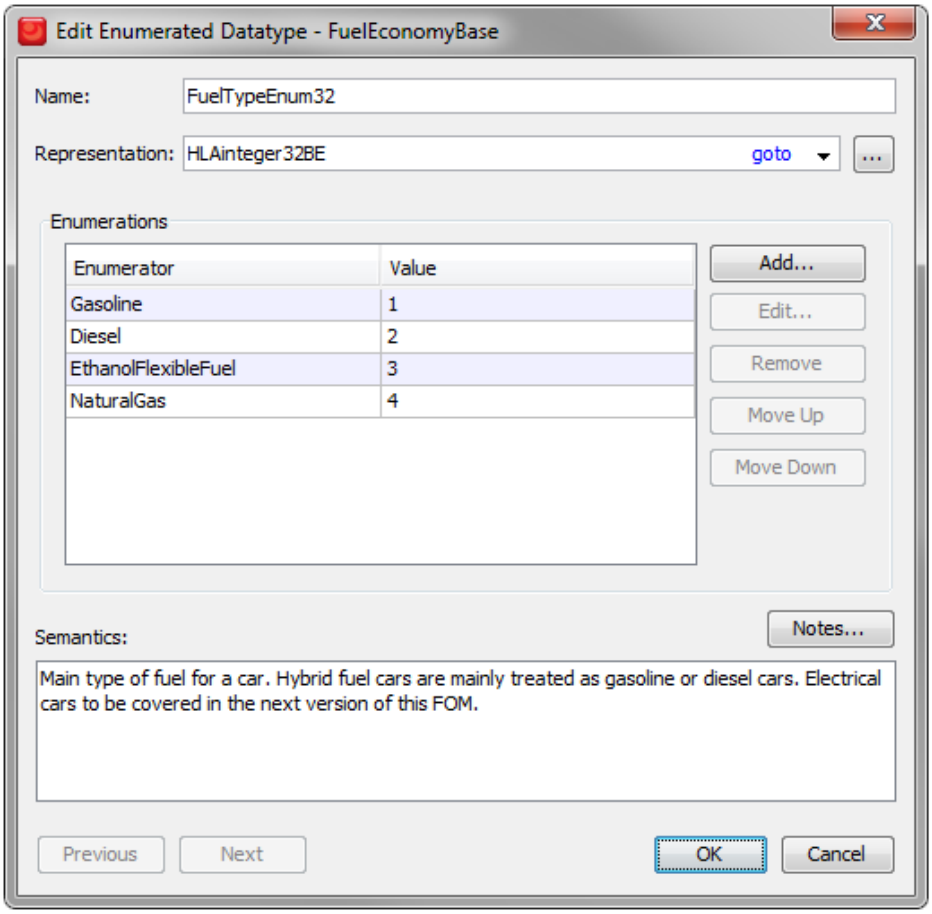


Figure 7-5: The Fuel Type enumerated value

This is an enumerated data type. It contains some important information:

- The **representation** of the enumerated value, typically an HLAoctet (8 bits), an HLAinteger16BE (16 bits) or an HLAinteger32BE (32 bits).
- The different **enumerated values** and their names.

Read more about Enumerated data types in section 4.13.5 of the Object Model Template Specification.

7.4. A closer look at the Position attribute

We will also look at the Position attribute.

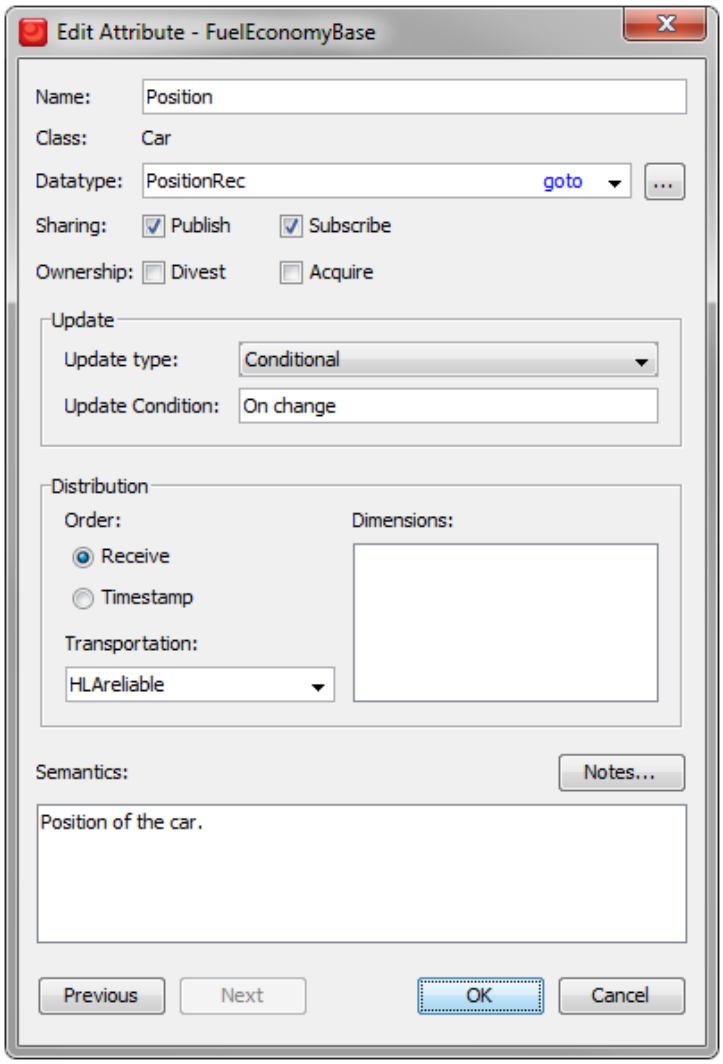


Figure 7-5: The Position Attribute

The position will change over time. The Update Type is Conditional and the condition is that an update is sent whenever the value changes.

The Data type is particularly interesting since it is a Fixed Record. Let's look at it:

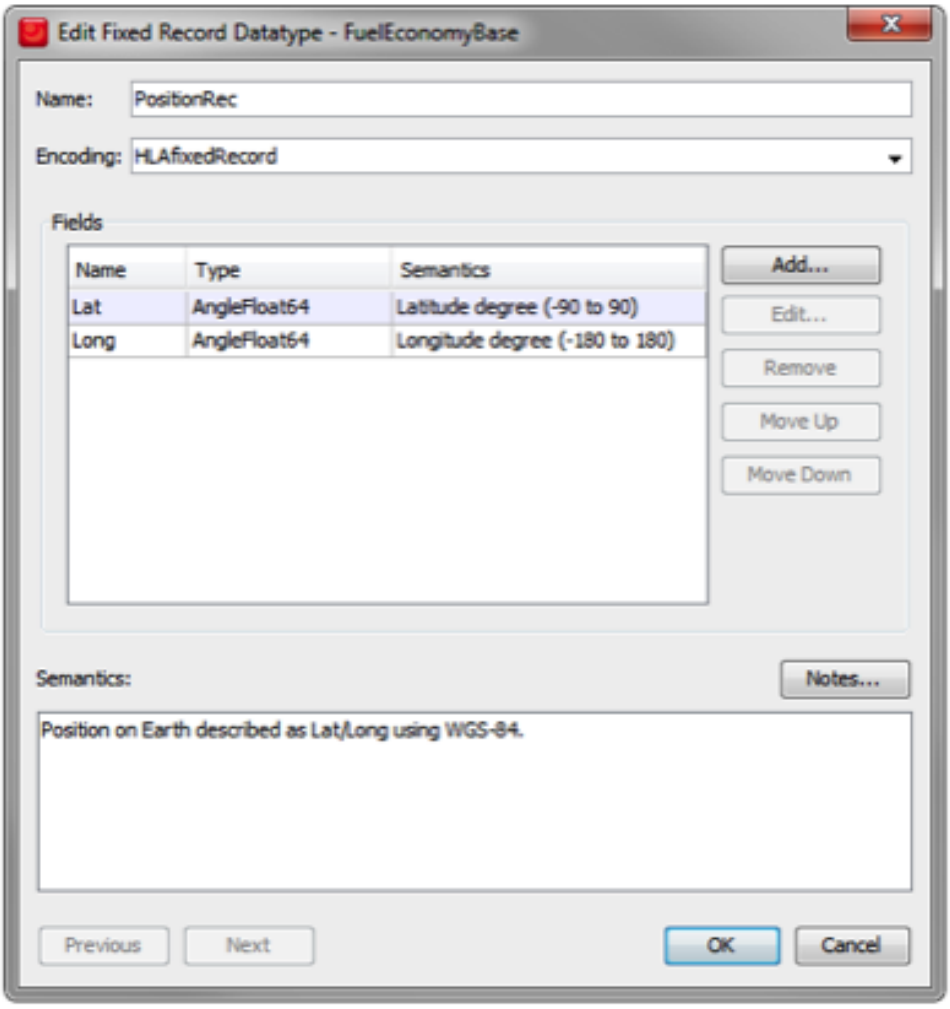


Figure 7-6: The PositionRec Fixed Record Data Type

It is important that we exchange data about both the Latitude and Longitude of a position at the same time so they are part of the same record. They both have the type AngleFloat64 so let's have a look at that:

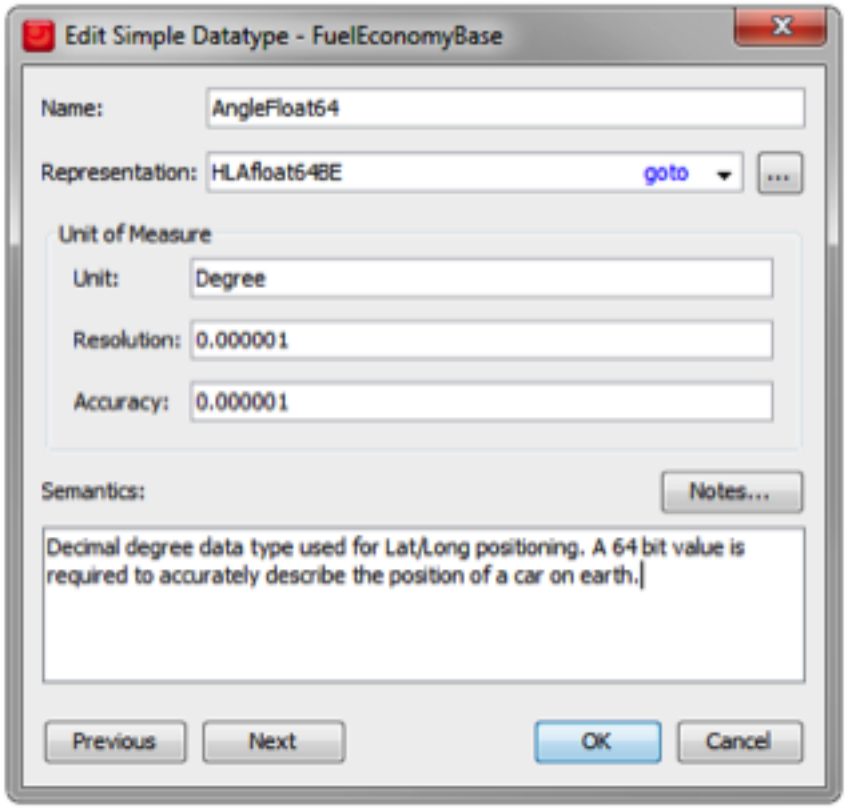


Figure 7-7: The Angle Simple Data Type

This is a Simple data type that is used for both Latitude and Longitude.

Read more about Fixed Record data types in section 4.13.7 of the Object Model Template Specification.

7.5. Practical Exercise

A lab for this chapter is provided in Appendix D-5

8. Objects – Calls for Registering and Discovering

- To be able to reference an object class we need to retrieve a handle for the class. We also need handles for the attributes.
- To be able to register and discover objects of a particular class as well as update/reflect attributes we need to publish and subscribe to it

8.1. Object Registration Overview

This chapter tells you how to register object instances of our Car class. We will also show how to discover Car instances that are registered by other federates. For now we will postpone how to update the attributes until the next chapter.

We will cover the following steps:

- Some initial preparations that need to be done.
- Registration of an object instance
- How to discover object instances
- Reserving a name for an object instance

8.2. Initial preparations for Registering and Discovering Objects

Initially we will need to do three things before we start the main simulation loop:

1. Define some variables
2. Get “handles”, which is a type of reference, for the object class
3. Publish and subscribe to the object class.

Here is the pseudocode:

```
VariableLengthData userSuppliedTag
AttributeHandleSet attrHandleSet
ObjectInstanceHandle carInstanceHandle
ObjectInstanceHandle carInstanceHandle2

carClassHandle = rti.getObjectClassHandle("Car")

nameAttrHandle = rti.getAttributeHandle(carClassHandle, "Name")
licensePlateNumber AttrHandle = rti.getAttributeHandle(carClassHandle,
                                                       "LicensePlateNumber")
fuelLevelAttrHandle = rti.getAttributeHandle(carClassHandle, "FuelLevel")
fuelTypeAttrHandle = rti.getAttributeHandle(carClassHandle, "FuelType")
positionAttrHandle = rti.getAttributeHandle carClassHandle, "Position")
attrHandleSet.insert(nameAttrHandle)
attrHandleSet.insert(licensePlateNumberAttrHandle)
attrHandleSet.insert(fuelLevelAttrHandle)
attrHandleSet.insert(fuelTypeAttrHandle)
attrHandleSet.insert(positionAttrHandle)

rti.publishObjectClassAttributes(carClassHandle, attrHandleSet)
rti.subscribeObjectClassAttributes(carClassHandle, attrHandleSet)
```

We will need an AttributeHandleSet when we specify a list of attributes. We will also create an empty userSuppliedTag.

We then need to fetch handles for the Car object class in the FOM as well as handles for the attributes of the Car. We insert the attribute handles into the attribute handle set. Finally we publish and subscribe to this object class. In this federation it will be the CarSims that publishes this object class and the MapViewer that subscribes to it, but this example shows both calls.

Here is some more information about these services.

8.3. HLA Service: Get Object Class Handle

This service returns a handle for the specified object class in the FOM. The name of a class may be described as "HLAobjectRoot.Car" or simply "Car", since you are allowed to omit the HLAobjectRoot. Note that this service (and many other Get Handle services) may throw a "not defined" exception meaning that there is nothing in the FOM that matches this string.

Read more about Get Object Class Handle in section 10.6 of the Interface Specification.

8.4. HLA Service: Get Attribute Handle

This service returns a handle for the specified attributes of an object. The object class handle needs to be supplied.

Read more about Get Attribute Handle in section 10.11 of the Interface Specification.

8.5. HLA Service: Publish Object Class Attributes

This service informs the RTI that the federate publishes the specified object class. This means that it can register object instances. It also informs the RTI that the federate can send updates for the specified attributes.

Read more about Publish Object Class in section 5.2 (TBD check) of the Interface Specification.

8.6. HLA Service: Subscribe Object Class Attributes

This service informs the RTI that the federate subscribes to the specified object class. This means that it wishes to receive notifications when a new object is registered by another federate. It also informs the RTI that the federate wishes to receive updates for the specified attributes.

Read more about Subscribe Object Class Attributes in section 5.6 (TBD check) of the Interface Specification.

8.7. A look at the RTI

By looking at the user interface of the RTI it is easy to verify that the federate has published the object class and attributes.

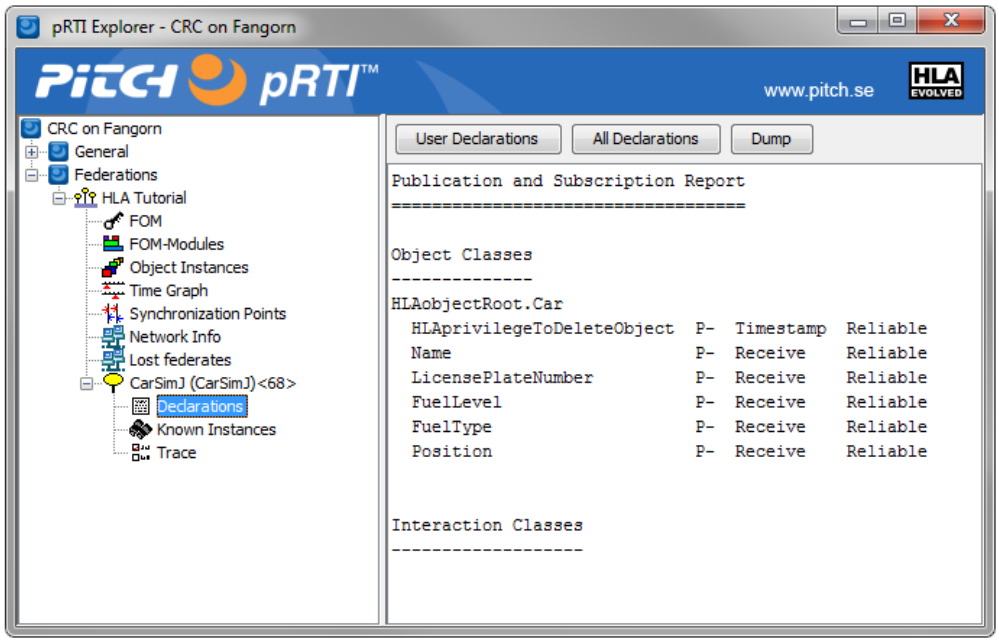


Figure 8-1: Publication of an object class with attributes as seen in the RTI GUI

8.8. Code for Registration of an Object Instance

To register a Car object instance we just need to make one call. Here is the pseudocode:

```
carInstanceHandle=rti.registerObjectInstance(carClassHandle)
```

This service registers a new object instance of the specified type. It returns a handle to the new instance. We want to save this handle so that we can update this object later on. This object will also get a name that is automatically generated by the RTI.

8.9. HLA Service: Register Object Instance

This service registers a new object instance. The RTI will create a unique instance name and a unique instance handle for the new instance. Optionally an instance name can be provided, which will be described later.

Read more about Register Object Instance in section 6.12 (TBD check) of the Interface Specification.

8.10. A look at the RTI

By looking at the user interface of the RTI it is easy to verify that the federate has registered the object instance.

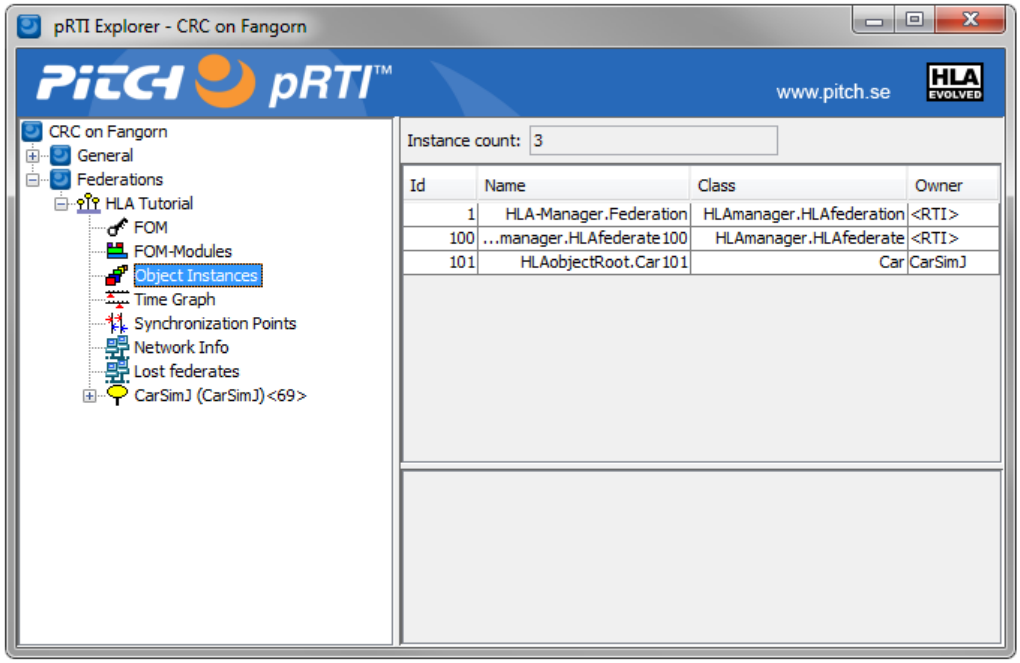


Figure 8-2: A registered object instance as seen in the RTI GUI

8.11. How to discover object instances

When a new object is registered by one federate, the RTI will make sure that it is discovered by other federates that subscribe to the specified class. If a federate joins a federation where there are already a number of objects, the RTI will also make sure the new federate discovers existing instances of a class it subscribes

to.

When an object is discovered a callback is made to the federate ambassador. We usually want to look at which class it is before further processing.

```
Method FederateAmbassador.DiscoverObjectInstance(theClassHandle, theObjectHandle,
theObjectName)

  IF theClassHandle=carClassHandle THEN
    HlaCar car = new HlaCar(theObjectName);
    _hlaCarMapping.put(theObjectHandle, car);
  END IF
```

It is really important to understand that the federate now knows about the existence of the new instance. But it still doesn't know anything about the fuel level or position of the car. It has no values for the attributes. It doesn't make sense to present this object to the main logic of you your simulation until the object is initialized, i.e. it has initial values for all relevant attributes.

8.12. HLA Service: Discover Object Instance (callback)

In this example we have simplified the parameter list. There are also optional parameters like a time stamp. Note also that there are actually two different versions of the Discover Object Instance method for the Federate Ambassador in the APIs. Different versions will be called depending on how many optional parameters that are present. It is highly recommended that you dispatch the calls that you don't use, in this case the ones with additional optional parameters, to the version of the service that you have actually implemented.

Read more about Discover Object Instance in section 6.9 of the Interface Specification.

8.13. More about object registration

In this example we have chosen to use object instances with automatic names since this is convenient, efficient and less error-prone. We will not use this instance name in our applications. Instead we will create our own attribute "Name" and use it to store a user-friendly name of each car.

In some cases you may want to specify the names of each object that you register. This can be done using an optional parameter. Before doing this you need to make a call to reserve the object names. The RTI will then reserve the object instance name across the federation, which may take some time in some RTIs. We recommend using the service Reserve Multiple Object Instance Names that allows you to register many objects at once. When the reservation is complete you will receive a Multiple Object Instance Name Reserved callback.

8.14. Practical Exercise

A lab for this chapter is provided in Appendix D-6

9. Objects - Calls for Updating and Reflecting Attribute Values

- The UpdateAttributeValues is used to send an update for a set of attributes for a specific object instance
- Updates are typically sent when the value has changed or when another federate needs and update.
- Your federate receives the ReflectAttributeValues with updated attribute values for attributes that is subscribes to

This chapter tells you how to work with attribute values for object instances. This includes both sending and receiving values. Before looking at any code we need to investigate a few aspects of updating attributes. We will start by looking at the FOM again.

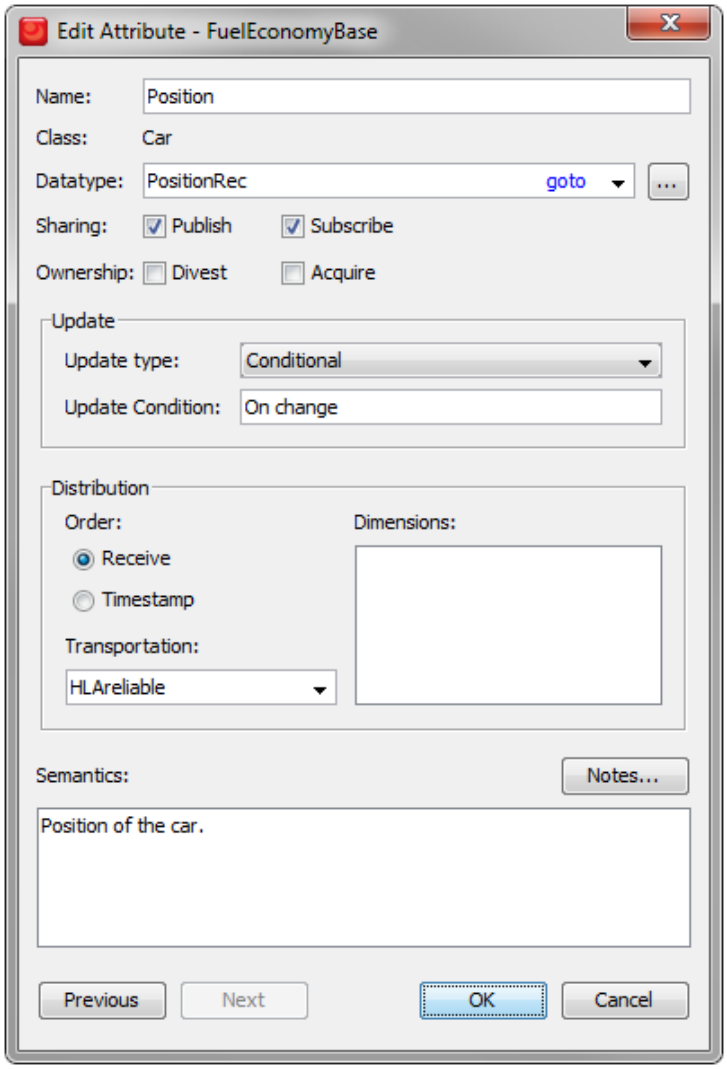


Figure 9-1: The position attribute revisited

The FOM describes three update types for an attribute:

- **Static**. A typical example is the name or registration plate number of a car. This value will be assigned and updated in the beginning of the simulation and it will not change.
- **Conditional**. A typical example is the position of a car which changes over time. One commonly used condition is “on change” which means that an update will be sent when there is a new value. Another possible condition is to send out a new value when it differs from the most recently sent update with a certain threshold.
- **Periodic**. This means that the value will be sent out periodically even if it did not change. We recommend using conditional/on change instead to save bandwidth and CPU.

There is some relation to the Transportation type used. We recommend using HLAreliable unless you have special requirements. The opposite is the HLAbesteffort transportation type where updates may be lost. In that case it makes sense to resend updates periodically to enable federates to catch up if they lost previous updates. The HLAbesteffort transportation type has some particular pros and cons since it can use a networking mode called “multicast”.

9.1. When to Update

We will now focus on the attributes with update type Static, like the registration plate number of a car and Conditional/On Change, like the position of a car. When should your federation send updates for such attributes? There are two cases:

1. When you have a **new value**. For Static attributes this means the first and only time when you have a new value for the name or registration plate number. For On Change attributes this means the first time you have a value and whenever you have an update to that value.
2. When another federate **requests a value**, no matter if this attribute has a Static or conditional update type. This typically happens when a federate joins into an already existing federation where there are already objects registered. This request normally happens automatically but your federate must respond correctly to it.

The same code for sending updated can be reused for this purpose but it needs to be called from two different places.

9.2. Initial preparations

We will show how to update the Position attribute of a Car, which is a record of Latitude and Longitude. We need to provide encoding helpers for this. We also need to define some variables for building and sending the update. Here is the pseudocode:

```
HLAfloat32BE latEncoder
HLAfloat32BE longEncoder
HLAfixedRecord positionEncoder

positionEncoder.addElement(latEncoder)
positionEncoder.addElement(longEncoder)

AttributeHandleValueMap attributes
VariableLengthData userSuppliedTag
```

9.3. Sending Attribute Updates

Here is the pseudocode. We have named this method "MySendCarPositionMethod" so we can show how to call it later on. We will simply set the new values, encode the data and send the update for the specified instance.

```
mySendCarPositionMethod (carInstanceHandle)
(
    latEncoder.Setvalue(38.897660)
    longEncoder.Setvalue(-77.036564)

    attributes[positionAttrHandle]=positionEncoder.encode()
    rti.updateAttributeValues(carInstanceHandle, attributes, userSuppliedTag)
)
```

Encoding is easy with the encoding helpers. Once you have built the correct structure you simply assign values for each field and the call the encode method for the structure.

9.4. HLA Service: Update Attribute Values

This service sends an attribute update for a particular object instance. The update contains a number of attribute/value pairs where the attribute is described by its handle. Some optional information can also be supplied, for example a User Supplied Tag which can contain metadata of the users choice.

Read more about Update Attribute Values in section 6.10 of the Interface Specification.

9.5. Responding to Provide

Imagine a federation where a new federate joins. If it subscribes to the Car object class it will now discover all Car instances. It will also need the most recent value for all attribute. The RTI can automatically ask your federate to provide it. This is done by calling the Provide Attribute Value Update callback.

```
Method FederateAmbassador.ProvideAttributeValueUpdate(theObjectHandle,
                                        TheAttributeHandleSet, theUserDefinedTag)

    theObjectClass=GetKnownObjectClassHandle(theObjectHandle)

    IF theObjectClass=carClassHandle THEN
                    IF theAttributeHandleSet.count(positionAttrHandle)!=0 then
                    mySendCarPositionMethod (theObjectHandle)
    END IF
```

In this method we first check which class the object belongs to. We then check if this is a request for the position value. If so we send the position. The code above is simplified. It is recommended that you send out one single update that contains values for all of the requested attributes for an object instance.

9.6. HLA Service: Provide Attribute Value Update (callback)

This callback is invoked on your federate when another federate requests the most recent value of an attribute of one of your object instances. This request is performed automatically by the RTI on behalf of a federate that discovers a new object instance (assuming the AutoProvide switch in the FOM is enabled).

Read more about the Provide Attribute Value Update in section 6.x of the Interface Specification.

9.7. Reflecting Attribute Values (callback)

To handle an incoming attribute update we need to decide which type of object it relates to, locate that object and then decode its attribute values. We already have encoders for this. Here is some pseudo code:

```
Method FederateAmbassador.ReflectAttributeValues(theObjectHandle, theAttributes)
   theObjectClass=GetKnownObjectClassHandle(theObjectHandle)

IF theObjectClass=carClassHandle THEN

HlaCar car = _hlaCarMapping.get(theObjectHandle);

   IF theAttributes.has(positionAttributeHandle) THEN
         myPositionDecoder.decode(theAttributes[positionAttributeHandle]);
         car.setPosition(myPositionDecoder);
   END IF
   IF theAttributes.has(fuelAmountAttributeHandle) THEN
         myFloat32Decoder.decode(theAttributes [fuelAmountAttributeHandle]);
        car.setFuelLevel(myFloat32Decoder);
   END IF

END IF
```

First we check which type of object this is. If it is a Car then we can fetch our HlaCar object that we created when we discovered the instance. Next we decode the attributes so that we can update the state of our car. We pass them to the decode method of the decoders. This code only updates a subset of the attributes of the car object. Note that we check if the update includes the attributes that we are interested in since not all attributes might be included. Note also that the decoders may throw exceptions if the incoming data is incorrect.

9.8. HLA Service: Reflect Attribute Values (callback)

There are also optional parameters like a time stamp. Note also that there are actually three different versions of the Reflect Attribute Values method for the Federate Ambassador in the APIs. Different versions will be called depending on how many optional parameters that are present. It is highly recommended that you dispatch the calls that you don't use, in this case the ones with additional optional parameters, to the version of the service that you have actually implemented.

Read more about Reflect Attribute Values in section 6.11 of the Interface Specification.

9.9. Practical Exercise

A lab for this chapter is provided in Appendix D-7

You are also encouraged to run the federation distributed across several computers as described in Appendix D-8.

10. Testing and Debugging Federates and Federations

- Federates should be tested as far as possible before they are brought to a federation
- Federates should first be tested standalone using the RTI GUI, by inspecting the data and by tracing the RTI calls
- A second step is to test with proven federates, proven data and to send the federate for certification.

So far we have been talking about how to develop federates for a federation. But how can you be sure that you got everything right before going to an integration event?

When developing one single program you are in control of all pieces of the software at the same time and can test and debug the entire code base. With several, loosely coupled and potentially distributed HLA federates it gets much more difficult. We will look at several approaches:

- Look at the RTI
- Inspect the data
- Inspect the RTI calls
- Test with proven federates
- Test with proven data
- Get your federate certified
- Test and understand the federate performance

Note that we focus on testing the interoperability aspects of a federate. Verifying the overall simulation capabilities of a federation is another story which is much more domain specific.

10.1. What should you test?

What you need to test varies from case to case. Assuming that the models in your federate have already been tested we need to look at least at the following things in the HLA module:

1. Are the publications correct?
2. Are the published objects and interactions indeed registered, updates and sent? At the right time?
3. Are the attributes updated at the right time with the correct values?
4. Is the encoding of the sent data correct?
5. Are the subscriptions correct?
6. Does the federate react to registered objects, updated attributes and received interactions as expected?

7. Does the federate gracefully discard received data that is obviously incorrect, for example attribute updates with incorrect data length?

Many federations usually require more than this. The federation agreement is usually a good place to start reading to understand these requirements.

10.2. Look at the RTI

This debugging technique has already been shown throughout this tutorial. The RTI user interface provides a lot of useful insight on what the HLA interface of your application has achieved so far. We have shown the following:

- You can check that your federate creates a federation execution and joins the federation as expected. You may also want to check that it joins with the expected IP addresses and other networking parameters.
- You can check that it resigns and potentially destroys the federation execution.
- You can inspect the declarations (publish/subscribe) of the federate.
- You can check that object instances get created and deleted as planned.

There is often considerably more information in the GUI of an RTI so you may want to take some time to go through it as well as reading the Users Guide of the RTI.

Note that the user interface and debugging features varies from RTI to RTI.

10.3. Inspecting Data with a Data Logger

You can use a data logger to record and inspect the data that you federate publishes. This enables you to scrutinize attribute updates and interactions. The data may be shown in hexadecimal or decoded form. Data loggers may be tailored for a particular FOM or may be able to use any FOM, for example an extended standard FOM.

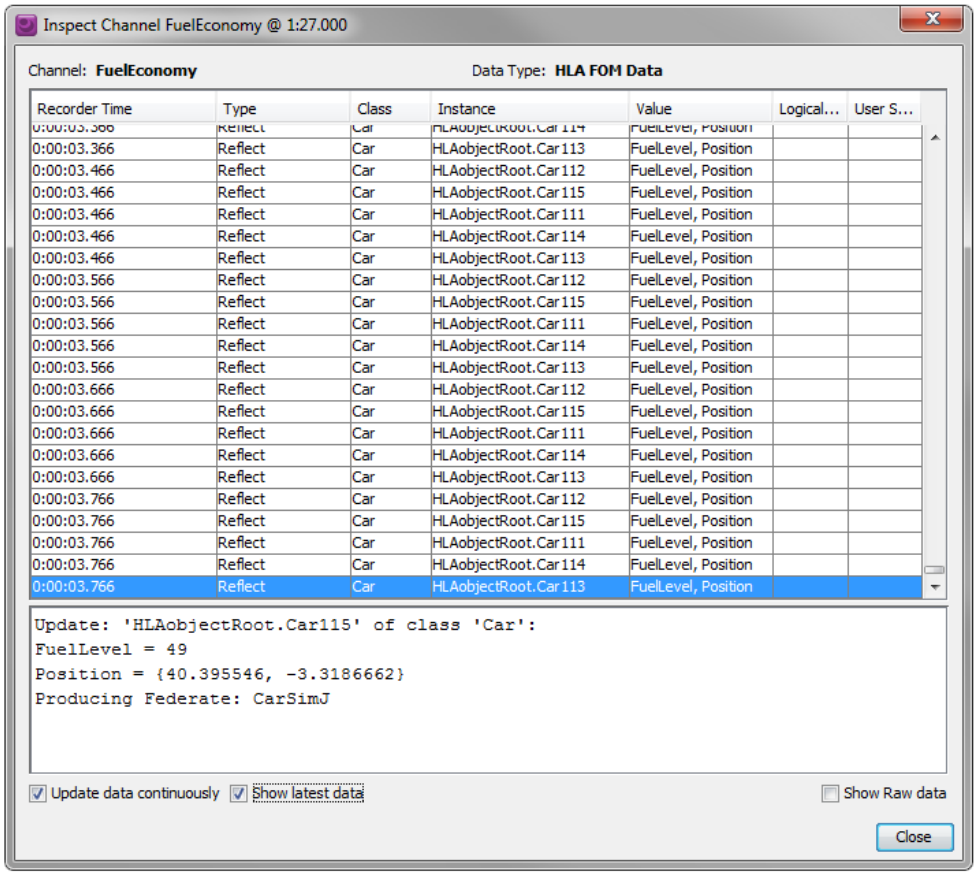


Figure 10-1: Logging data from the Fuel Economy Federation

Figure 10-1 shows a data logger that captures HLA FOM data, in this case Reflect Attribute Values, decodes the data and shows it on screen.

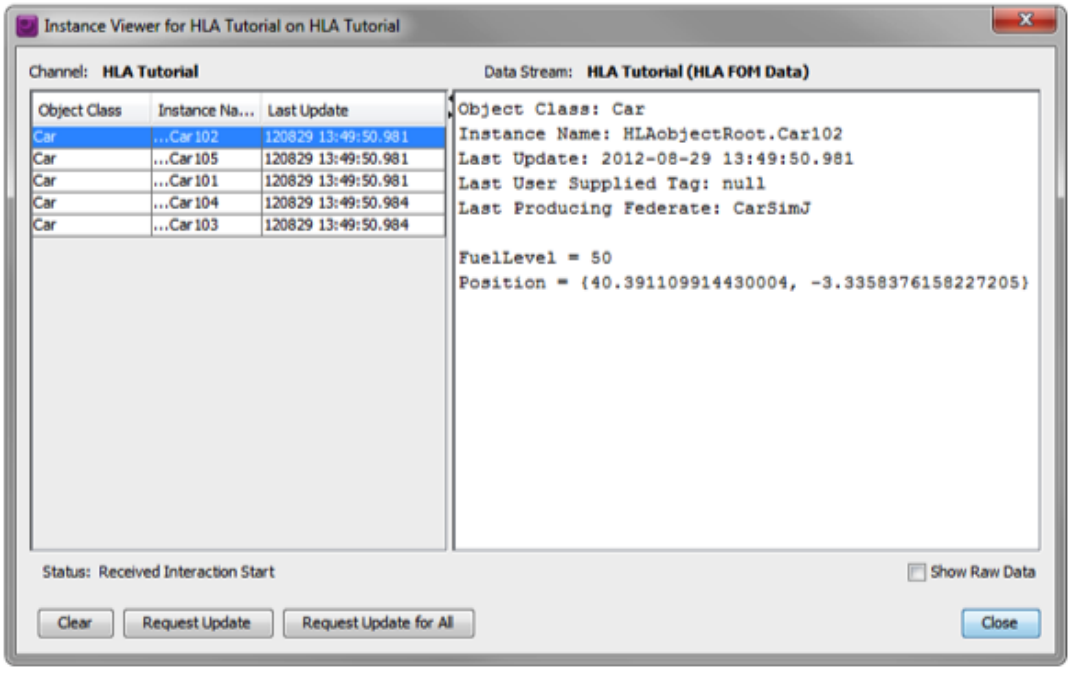


Figure 10-2: Instance viewer of a data logger

Another useful way to present the logged data is using an instance view, as shown in Figure 10-2.

10.4. Tracing RTI Calls

To perform a detailed inspection on how you federate interacts with the RTI you may trace the RTI calls. This can be done in some debuggers. Some RTIs also provide the ability to print traces. Here is an example of a printout of the RTI calls for the Fuel Economy federation.

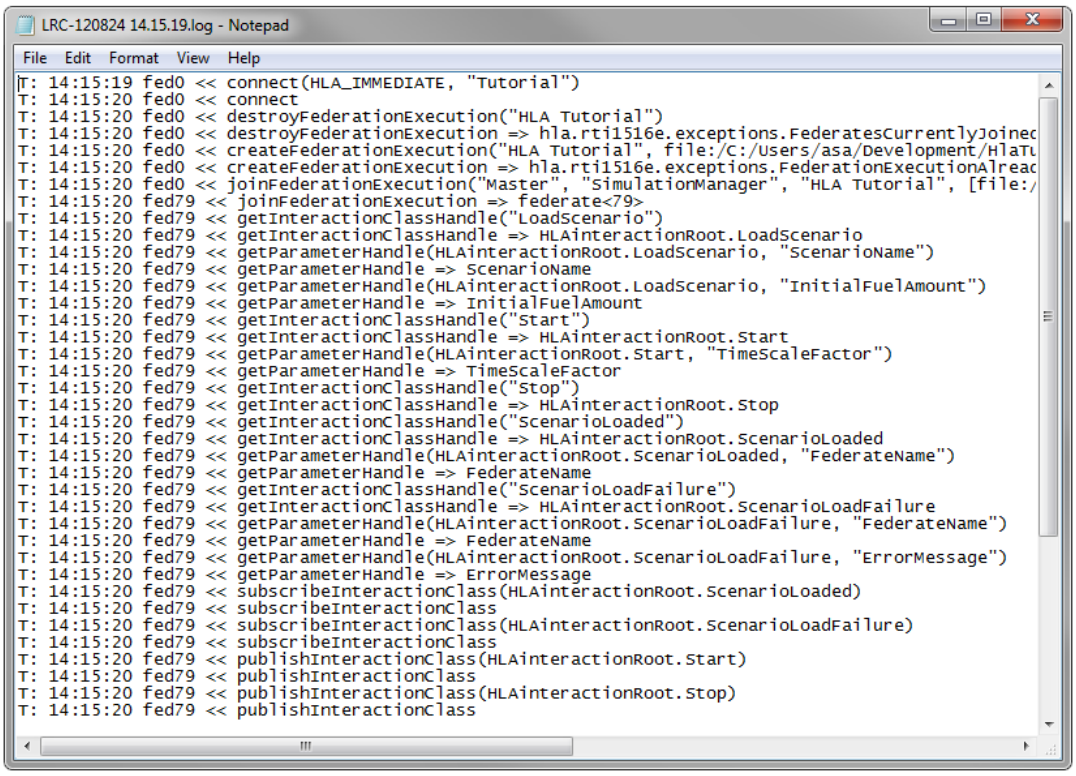


Figure 10-3: A Fuel Economy trace example from pRTI

In many cases you may want to focus on one aspect of the services, for example how object instances are registered. This is also important for full-scale simulations where you may have thousands and thousands of attribute updates and interactions. Here is an example of how to focus on certain services:

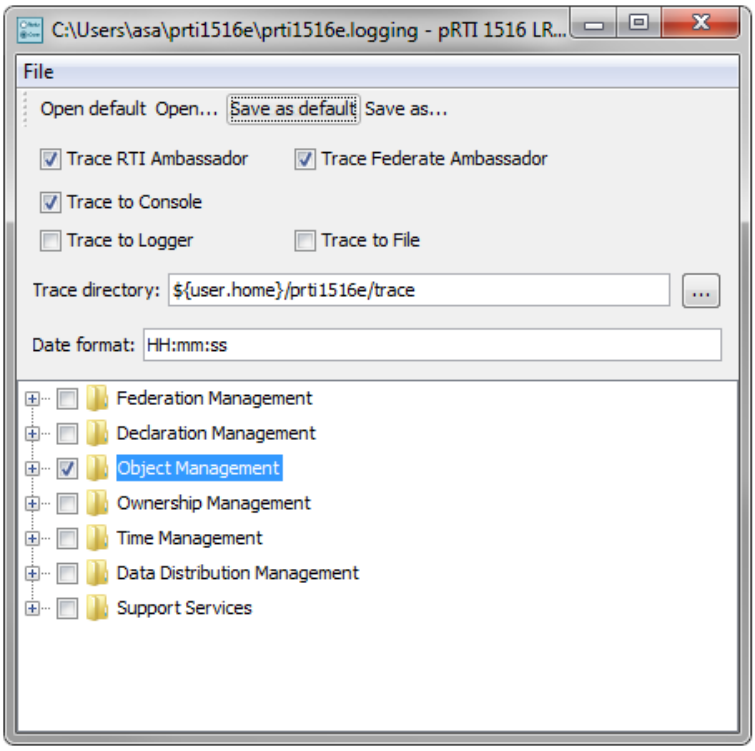


Figure 10-4: Selecting HLA services to trace

10.5. Test with proven federates

One obvious way to test your federate is to see what happens when you connect it to other, proven federates. This is indeed a good test that can reveal a number of issues that you didn't think of when developing you federate. It may also be used to discover requirements that do exist but haven't been well documented.

This approach does have a few drawbacks:

- It is difficult to know exactly how much of the services and the FOM data exchange that you actually test. You may know very little about if and how a receiving federate processes data from your federate.
- It is difficult to discover if data was correctly exchanged if you don't know the functionality of both federates very well.
- Both federates may be using the same, incorrect interpretation of the FOM, in particular if they were developed in the same project or by the same organization.

10.6. Test with proven data

In larger projects you can usually get access to proven and correct data, usually collected using a data logger. This data can be used to stimulate your federate. Logged data is highly useful for automated testing.

If you develop a number of new federates you may use data loggers to exchange data between teams to verify compatibility.

10.7. Get your federate certified

Several government organizations provide certification of HLA federates, in particular for defense oriented federates. This is a good way to get your federates tested and to get an official proof of HLA compliance.

In order to get your federate certified you need to provide a specification of what HLA services that is uses and what the federate publishes and subscribes to. The federate certification will then verify that the federate behaves as specified. The certification only tests the interoperability aspects. It does not test the accuracy of your simulation models, for example if the aerodynamics model in a flight simulator is correct.

10.8. Test and understand the federate performance

Every federate has a performance impact on the federation. Every federate also has some performance limitations that may impact the entire federation. The limiting factor in most federations is how many updates and interactions that are exchanged. Two important questions are:

1. How many updates/interactions are sent per second from your federate in a typical scenario?
2. How many incoming updates/interactions per second can your federate handle?

In most cases the ability to process incoming updates will be the limit for the entire federation. Consider a federation where ten federates send 1000 updates each per second and where all federates subscribes to everything. The biggest problem is obviously not to send 1000 updates per second but to process 9000 incoming updates per second.

11. Object Oriented HLA

- Object Oriented HLA gives convenient and type-safe access to the shared information using proxy objects
- OO-HLA is a design approach for middleware rather than a standard
- OO-HLA middleware can be developed by manual programming or code-generation from a FOM

11.1. What is Object Oriented HLA

Object Oriented HLA is not a standard. It is a design pattern for the HLA Module of an application. It builds upon the concept of FOM-specific proxy objects. The HLA Module typically works in two ways:

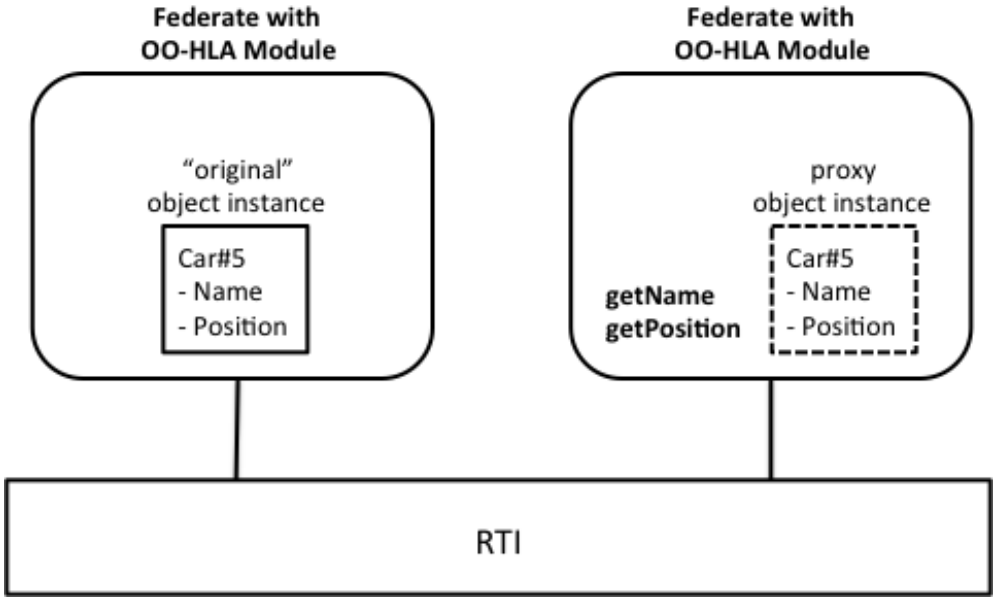


Figure 11-1: Get Value from Proxy object

If another federate registers an HLA object instances, for example of the class Car, then the HLA Module presents a C++ or Java object that corresponds to that instance. The attributes of that object are populated as updates are received. To get these values you use a type-safe "get" operation, for example Car.getName. This can technically be seen as a remote or proxy object.

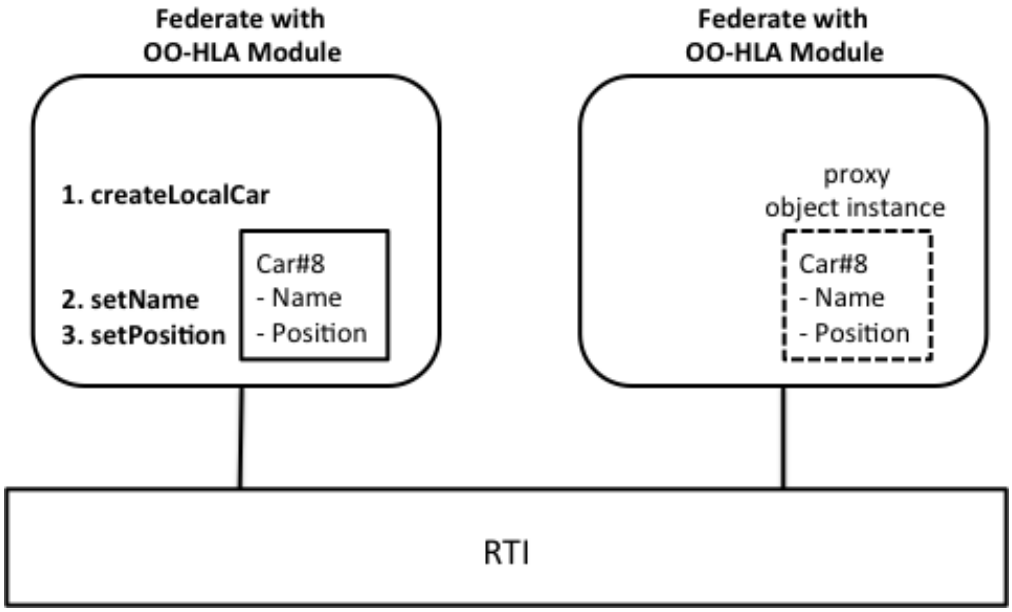


Figure 11-2: Set Value on Local object

Your federate can create your own "local" object instances. The HLA module will then register the object in the federation. The attributes of that object can be updated with type-safe "set value" operations. These attribute values will automatically be sent to other federates.

Some of the advantages of object oriented HLA are:
- Fast and easy to add an HLA module to your simulator
- You don't need to learn all the details of HLA
- Reduced risk of incorrect data encoding and decoding and other HLA programming mistakes.

Some of the disadvantages are:
- The middleware is usually locked to one particular FOM. It is not suitable for general-purpose tools that need to handle different FOMs, for example flexible data loggers.
- Need to acquire or develop the Object Oriented HLA module.
- Need to be careful with the configuration so that it does not subscribe to more object classes than necessary, which may impact performance.

11.2. Types of OO-HLA implementations

There are several types of object oriented HLA modules. The most obvious approach is to program your own HLA module. This assumes that you know HLA reasonably well. This module can then be shared with your co-workers who need to understand less about HLA.

You may also buy a commercial HLA module that implements some particular FOM of interest, for example the RPR FOM. These may even implement other interoperability standards, like DIS in parallel with HLA.

A third approach is to use a code generator that takes a FOM as input and generates an object oriented HLA module for this FOM. The resulting code is then integrated into your federate project. You can develop your own code generator or get a code generator product.

11.3. Using OO-HLA in the Fuel Economy Federation

We will now look at a sample object oriented HLA code generator named Pitch Developer Studio. First we will see how to generate the code and then we will look at how to integrate the code into our project. We start by loading the FOM into the tool and select the Car object class. We select that we want to support both local and remote object instances. We then check all attributes of the Car class.

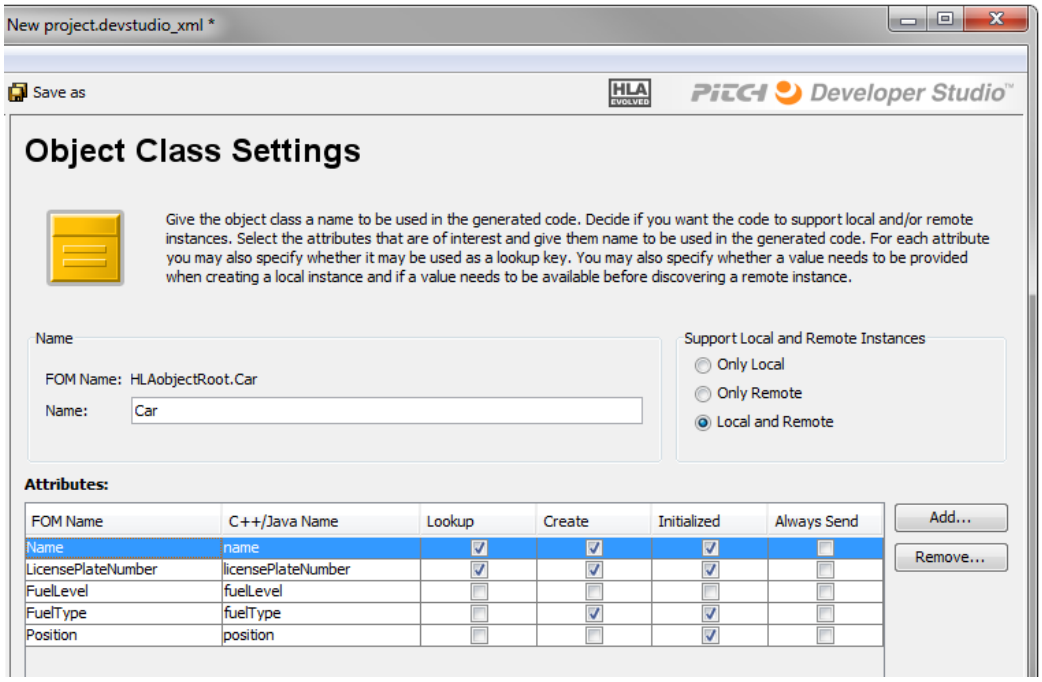


Figure 11-1: Attribute properties in Pitch Developer Studio

This product introduces a few additional features for attributes, for example:

Lookup: The generated code will contain lookup functions so that we can easily find an instance. By checking this box we indicate that this is an attribute that we want to use as a lookup key. In this case we select Name and LicensePlateNumber as lookup keys.

Create: The value of this attribute shall be supplied when a new instance is created. This is something we want to use for Name, LicensePlateNumber and FuelType.

Initialized: It may not be useful for your program to discover remote object instances that have attributes that has not yet received any updates, i.e. have no values. By checking this box we make sure that the HLA module will not present

the instance to your program until it is sufficiently initialized. For the car class we check all of the attributes except for the fuel level.

After setting some more parameters, like package name and source code directory, we can now generate code in C++ or Java.

11.4. Sample OO-HLA program code

The generated code has some important classes:

HlaWorld is a class that handles the connection to the federation and provides methods like Connect and Disconnect. Note that these in turn will call HLA services like Connect, Create Federation Execution and Join etc. This is taken care of automatically so we don't need to worry about this.

CarManager is a class that handles all local and remote car instances as well as providing additional functionality for the HLA Car class.

Here is some sample code that connects to the federation, registers a new car instance and finally disconnects from the federation. Note that this requires a developer to write only a fraction of the number of lines of code shown in previous chapters.

```
HlaWorld _hlaWorld = HlaWorld.Factory.create();
_hlaWorld.connect();

HlaCarManager _carmanager=_hlaWorld.getHlaCarManager().
HlaCar _car = _carManager.createLocalCar("Sample Car", “ABC123”, FuelTypeEnum.DIESEL);

// Simulate

_hlaWorld.disconnect()
```

Here is some sample code that updates the fuel level and position of the car. Note that it uses an object called an Updater that enables us to update several attributes in one atomic transaction. Also note that these calls provides type safety. You cannot accidentally send a string value for the fuel level without getting a compile time error.

```
HlaCarUpdater myUpdater = car.getHlaCarUpdater();
myUpdater.setFuelLevel(13);
myUpdater.setPosition(PositionRec.create(23.451, 45.662));
myUpdater.sendUpdate();
```

Finally, here is some code that loops through all cars in the federation and prints out their name and fuel level.

```
for (HlaCar car : _hlaWorld.getHlaCarManager().getCars())
{
    print("Car " + car.getName() + " has fuel level " + car.getFuelLevel());
}
```

The above is just a sample of object-oriented HLA. Different products use different approaches. A federate that uses object oriented HLA looks just like any other federate when used in a federation. You can freely mix federates that use object oriented HLA with traditional federates.

12. Summary and Road Ahead

This chapter provides a summary and some deeper insights into federations and interoperability. It also describes some additional HLA services that will be covered in Part 2 of the tutorial.

12.1. About the Fuel Economy Federation – interoperability and reuse

We have built a federation that demonstrates and explains the basic capabilities of HLA in detail. We have seen how the federates interoperate based upon a Federation Agreement. We have also seen how the data exchange follows the FOM that we developed. We have been able to register and discover object instances and update their attribute. We have sent and received interactions.

There are also a number of more advanced aspects that you may also have observed:

- Since we can add more publishers and subscribers, without modifying existing federates, it is easy to gradually extend the federation. We can also reuse federates in new federations.
- We can also easily replace federates and, for example, introduce another publisher of simulated cars.
- The targets of interactions are not hard coded in the sender federate. Because of this we can add more federates that react to an event. It is thus easy introduce data loggers and visualizers.

A high degree of interoperability and reuse has been achieved between simulations that follow the Fuel Economy Federation Agreement. Still we have imposed very few constraints on the internal architecture and implementation of each participating federate.

12.2. DSEEP - a process for developing federations

In this tutorial we have looked at the technical development of a federate for a federation with a federation agreement and a FOM. But how does this happen in real life? How do you arrive at a design and make sure that you can deliver a working federation on time? There is a process for design, development and execution of distributed simulations called DSEEP. It consists of seven steps.

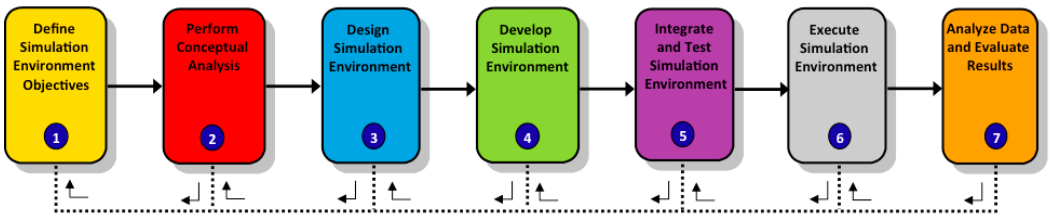


Figure 12-1: DSEEP Process for Simulation Interoperability

The steps are as follows:

1. Define Federation Objectives and constraints. In this step we specify the goals and any constraints, like time or budget
2. Perform Conceptual Analysis. In this step we develop a typical scenario for the simulation and produce a conceptual model.
3. Design Federation. In this step we select federates, allocate responsibilities to them and decide on any development of new federates
4. Develop Simulation Environment. In this step we develop the Federation Object Model (FOM), the Federation Agreement and adapt or develop federates
5. Integrate and Test Simulation Environment. In this step we perform testing, sort out bugs and validate that the federation works as intended.
6. Execute Simulation. In this step we finally execute the simulation.
7. Analyze Data and Evaluate Results.

DSEEP is also an IEEE standard (1730-2010) which is freely available to SISO members. It can also be purchased on the IEEE web site.

12.3. More HLA Features: Ownership, Time Management and DDM

In Part Two of the tutorial we will take a look at a number of additional HLA features. These are:

Ownership. What if we want one of the cars that is currently simulated by CarSimC to be simulated by CarSimJ instead? What if we want to introduce a separate federate that only performs fuel level calculation for the cars that are modeled by the CarSims? What if we want to have a failover simulator that takes over the responsibility for modeling cars if any of the regular car simulators fail? This can be achieved with HLA Ownership Management.

Time. May readers have probably noticed that the handling of scenario time is less than satisfactory in the current federation design. Network delays may cause the federates to start at different points in time. Different hardware clocks may cause the scenario time to run at different speed on different computers. HLA has some very powerful services that enable you to exchange time stamped data and to coordinate the time advance of your simulations.

Data Distribution Management. Imagine that we have thousands of cars. How can we filter out just a limited number of these on different criteria like fuel type, position or something else? The answer is DDM services, which becomes more interesting as your federation and scenario grows.

Management Object Model (MOM). It is possible to gain a deeper insight into which federates that have joined and their state by asking the RTI. This is very useful for example in the Master federate.

More. We will also look at fault tolerance, advanced FOM development , encoding helpers and more in Part Two.

12.4. Have Fun!

Before you move on to Part Two we would like to encourage you to play around with the samples. Implement some code of your own. Study the standard. Visit the SISO web site and read about other peoples experience with HLA.

But most important: go back to your original simulations and start thinking about how HLA can help you build simulations that do things that you couldn't do before. Solve problems! Be innovative!

We hope to see you soon for the second part of the HLA Tutorial.

Appendix A: The Fuel Economy Federation Agreements

1. General

Purpose, audience

The Fuel Economy Federation is an analysis federation that is intended to study and compare the fuel consumption of vehicles under different conditions. This federation agreement is the design specification (or contract) for the federation. It provides requirements for the interoperability aspects of any participating federate.

This Federation Agreement is a sample federation agreement developed for learning purposes. The intended audience is anyone who wants to understand and potentially develop federates that need to participate in the Fuel Economy federation. It can also be used as a basic template for small federation agreements. Note that the SISO FEAT group provides a more advanced template for federation agreements.

Revision history

Version	Date	Author	Descriptions
1.0	2012-07-01	BM	First complete version

Abbreviations, definitions

FOM	Federation Object Model
HLA	High-Level Architecture as defined in IEEE 1516-2010
IEEE	The Institute of Electrical and Electronics Engineers, Inc.
FEAT	Federation Agreement Template, a working group within SISO
SISO	Simulation Interoperability Standards Organization
TBD	To Be Done

References

1. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA), IEEE 1516-2010, www.ieee.org
2. Fuel Economy Federation FOM, see appendix B
3. Description of Federates and File Formats, see Appendix C

Related documents

- none -

2. Overview

Conceptual domain model

The following entities and processes are simulated in this federation:

- Cars and their fuel consumption when driving along a specified track. Both diesel and petrol cars are included.

Federation and participating federates

The federation can have three types of federates:

1. The **Master** federate is responsible for the management of the execution, for example assigning scenarios, starting and stopping. There shall be exactly one Master federate in each federation.

2. **Car Simulator** federates are responsible for modeling car objects, their movements and fuel consumptions. There may be any number of car simulator federates. Each federate may model one or more car objects.

3. **Analysis, visualization and data collection** federates. These are used for analysis of the simulation results during and after the execution. They may only consume data. They may not affect the simulation. There may be any number of these federates.

In the first version of the federation the name of the federation execution shall be “Fuel Economy”. The following federate names shall be used.

Federate Name	Description
Master	Master federate that manages the federation
CarSimC	Simulates A-Brand cars. Written in C++
CarSimJ	Simulates B-Brand cars. Written in Java
MapViewer	Displays cars on a map

FOM

The Fuel Economy Federation FOM (FEF FOM) is provided in Appendix B.

Overview of information flow, DDM usage (optionally)

Information about car objects will be produced by the car simulators and consumed by analysis, visualization and data collection federates.

Management interactions will be produced by the Master federate and responded to by the car simulators.

Relation to other systems

No relations.

3. General agreements

Standards, hardware, software and networks

Federates shall run on one or more computers with Windows 7, RedHat Enterprise Linux 6 or Mac OS X 10.6 using their native TCP/IP networking. A

network with at least 100 Mbps bandwidth using a 100 Mbps (or better) switch shall be used. The end user decides the network addresses for participating computers.

The native HLA 1516-2010 services and APIs shall be used by all federates. An RTI for HLA 1516-2010 from Pitch, version 4.4 shall be used.

Principles for sending interactions

All interactions shall be sent with the reliable transportation type.

Principles for updating attributes and ownership

Updates for static attributes shall be sent reliably on the creation of the object and upon request.

Updates for attributes that change over time shall be sent reliably whenever a new value is available (i.e. on change).

Federates shall implement the Provide Attribute Values callback for all attributes in order to support later joining federates to get the most recent value.

The federation shall have the AutoProvide switch enabled in the FOM. Federates shall thus not be required to call the Request service when it discovers a new object instance from another federate.

Ownership services are not used in this federation.

Technical representation of data

A full specification of the data representation is available in the FEF FOM in Appendix B. All data types are based on the standard Basic Data Types in the HLA 1516-2010 standard.

Strings shall be exchanged using the HLAunicode representation.

Integers and floats shall use Big Endian encoding.

Enumerated values shall use HLAinteger32BE representation.

Time management and time

This federation does not use HLA Time Management services.

The scenario time shall be set to zero when a scenario has been successfully loaded. The start interaction shall start the scenario time running. The federation then runs in “real” time multiplied with a time scale factor. If this factor is 1.0 then one second in the scenario time corresponds to one second of real time, i.e. real time. If, for example, the value is 15 then the scenario time shall run 15 times faster than real time.

The stop federation shall stop the scenario time. After a stop interaction has been received the start interaction shall start the time at the current scenario time value.

No time stamps are exchanged during the execution. Each federate may use the internal time representation that meets their needs.

4. Exchange of information

Information about Car objects

Updates shall be sent for all attributes of Car objects whenever the value changes.

Management interactions

These are specified in the section Managing the federation.

5. Managing the federation

Start-up and shutdown

Federates are started and shut down manually. The end user of the federation is responsible for verifying that all required federates are available before starting up a scenario. No federates are allowed to start after the scenario management has taken place.

Overview of scenario and execution management

The scenario and execution management follows the following pattern:

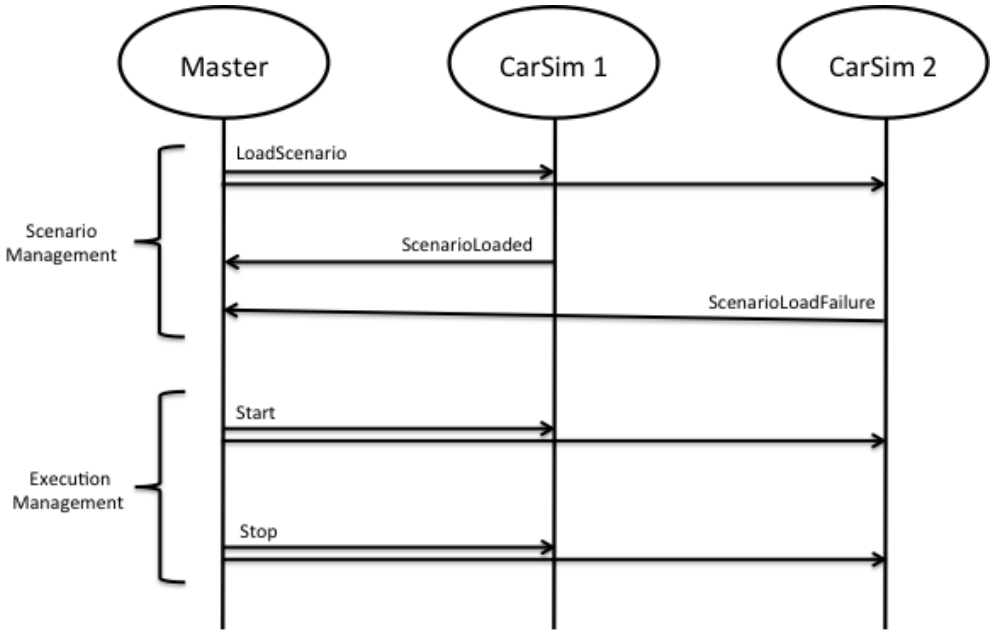


Figure A-1: Scenario and execution management

The scenario management is required to take place before the execution management. The Master federate sends a LoadScenario interaction. Each federate responds, indicating whether the scenario was successfully loaded or not. The operator of the Master federate is responsible for manually assessing whether the execution should be started, based on the responses.

The execution management is performed as follows: The Master initially sends a start interaction. Finally it sends a stop interaction. The end user is responsible for verifying that participating federates have correctly reacted to these interactions.

Scenario management

The Master federate is responsible for coordinating the scenario. The scenario is stored in a separate file as described in the appendix “Description of Federates and File Formats”. Three interactions are used for coordination of the common scenario:

Interaction: LoadScenario (ScenarioName, InitialFuelAmount)
This interaction shall be published by the Master federate and subscribed by all Car Simulators and Map Viewer. The name of the scenario and the initial fuel amount shall be supplied. Each federate shall load the specified scenario when it receives this interaction. Car simulators shall also assign the specified amount of fuel to each car that it simulates. The scenario time of all federates shall be set to zero. Note that this interaction is not allowed when the simulation is running.

Interaction: ScenarioLoaded(FederateName)
This interaction shall be published by each Car Simulator and subscribed by the Master federate. Each Car simulator shall send this interaction when it has successfully loaded a scenario. Other federates, like the Map Viewer, may optionally send this interaction. The name of the federate shall be supplied. The Master shall present the information received to the user.

Interaction: ScenarioLoadFailure(FederateName, ErrorMessage)
This interaction shall be published by each Car Simulator and subscribed by the Master federate. Each Car simulator shall send this interaction it fails to load a scenario. Other federates, like the Map Viewer, may optionally send this interaction. The name of the federate and an error message shall be supplied. The Master shall present the information received to the user who is responsible for taking appropriate action.

Execution management (starting and stopping)

The Master federate is responsible for coordinating the start and stop of the simulation. Two interactions are used:

Interaction: Start(TimeScaleFactor)

This interaction shall be published by the Master and subscribed by each Car Simulator and Map Viewer. Each Car Simulator shall start simulating when this interaction is received. If the Car Simulator has stopped simulating due to a Stop interaction the Start interaction shall cause the federate to continue simulating at the current time. The TimeScaleFactor indicates the ratio between the elapsed scenario time and the real time. See the section about time (above).

Interaction: Stop
This interaction shall be published by the Master and subscribed by each Car Simulator and Map Viewer. Each Car Simulator shall stop simulating when this interaction is received. The stop shall be performed in such a way that the simulation can be restarted if a Start interaction is received later on.

Note that the above interactions are used to start and stop the scenario time, i.e. the entire simulated world. The purpose is not to start and stop cars.

Save/restore

Not used in this simulation.

Error handling

Errors shall be handled as follows:

1. **Federate Internal Errors**: If an internal error occurs within a federate that prevents it from continuing to perform its responsibilities it shall signal the error on the console and then resign from the federation.

2. **Incorrectly encoded data in updates and interactions:** If incorrectly encoded data is received by a federate then this data shall be reported to the user and discarded. It is required for all federates to be able to detect at least incorrect data size for incoming data.

3. **Incorrect sequence of management interactions:** If a management interaction occurs that is out of sequence then this shall be reported to the user and discarded. An example of this is sending a Start interaction when no LoadScenario interaction has been sent.

4. **Incorrect set of federates:** It is up to the user to detect and handle any errors in the set of federates, for example missing Master federate or zero Car Simulators.

6. Handling output

Logging

Data shall be collected using a COTS data logger for HLA. This is optional.

Analysis

Analysis shall be performed by importing and processing logged data into MS Excel.

7. Federation specific section

- Not used -

8. Appendices

See Appendix B for the Fuel Economy Federation FOM.
See Appendix C for File formats.

Appendix B: The Fuel Economy FOM

This appendix provides the FOM in the standard table format specified in the HLA Object Model Template. The actual file format is based on XML. The Fuel Economy FOM is provided in XML format as part of the "HLA Evolved Starter Kit" and can be opened using a standard text editor, an XML editor or a dedicated HLA OMT Tool.

Identification table

Category	Information
Name	Fuel Economy FOM
Type	FOM
Version	1.0
Modification Date	201208-28
Security Classification	Unclassified
Purpose	Tutorial FOM
Application Domain	Engineering
Description	FOM for the Getting Started with HLA Evolved kit
POC	
POC Type	Primary Author
POC Name	Björn Möller
POC Organization	Pitch Technologies
POC Telephone	+46 13 13 45 45
POC Email	info@pitch.se
POC	
POC Type	Contributor
POC Name	Åsa Wihlborg
POC Organization	Pitch Technologies
POC Telephone	+46 13 13 45 45
POC Email	info@pitch.se
References	
Document	The HLA Tutorial (Sep 2012)
Glyph	
Type	JPEG
Alt	Fuel pump
Height	36
Width	36

Object class table

HLAobjectRoot (N)	Car (PS)

Interaction class table

HLAinteractionRoot (N)	LoadScenario (PS)
	ScenarioLoaded (PS)
	ScenarioLoadFailure (PS)
	Start (PS)
	Stop (PS)

Attribute Table

Object	Attribute	Data type	Update type	Update condition	D/A	P/S	Available Dimensions	Transportation	Order
Car	Name	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	LicensePlate Number	HLAunicodeString	Static	NA	N	PS	NA	HLAreliable	Receive
	FuelLevel	FuelInt32	Conditional	On change	N	PS	NA	HLAreliable	Receive
	FuelType	FuelTypeEnum32	Static	NA	N	PS	NA	HLAreliable	Receive
	Position	PositionRec	Conditional	On change	N	PS	NA	HLAreliable	Receive

Parameter Table

Interaction	Parameter	Datatype	Available dimensions	Transportation	Order
LoadScenario	ScenarioName	HLAunicodeString	NA	HLAreliable	Receive
	InitialFuelAmount	FuelInt32			
ScenarioLoaded	FederateName	HLAunicodeString	NA	HLAreliable	Receive
ScenarioLoadFailure	FederateName	HLAunicodeString	NA	HLAreliable	Receive
	ErrorMessage	HLAunicodestring			
Start	TimeScaleFactor	ScaleFactorFloat32	NA	HLAreliable	Receive
Stop	NA	NA	NA	HLAreliable	Receive

Switches Table

Switch	Setting
Auto Provide	Enabled
Convey Region Designator Sets	Disabled
Convey Producing Federate	Disabled
Attribute Scope Advisory	Disabled
Attribute Relevance Advisory	Disabled
Object Class Relevance Advisory	Disabled
Interaction Relevance Advisory	Disabled
Service Reporting	Disabled
Exception Reporting	Disabled
Delay Subscription Evaluation	Disabled
Automatic Resign Action	CancelThenDeleteThenDivest

Simple Datatype Table

Name	Representation	Units	Resolution	Accuracy	Semantics
FuelInt32	HLAinteger32BE *[FuelEconomyBase_1]	Liters	1	1	Integer that describes a fuel volume
AngleFloat64	HLAfloat64BE	Degree	0.000001	0.000001	Decimal

					degree data type used for Lat/Long
ScaleFactorFloat32	HLAfloat32BE	NA	0.001	0.001	Used for Time Scale Factor

Enumerated Datatype Table

Name	Representation	Enumerator	Values	Semantics
FuelTypeEnum32	HLAinteger32BE	Gasoline	1	Main type of fuel for a car.
		Diesel	2	
		EthanolFlexibleFuel	3	
		NaturalGas	4	

Fixed Record Datatype Table

Record Name	Field			Encoding	Semantics
	Name	Type	Semantics		
PositionRec	Lat	AngleFloat32	Latitude degree (-90 to 90)	HLAfixedRecord	Position described as Lat/Long using WGS-84
	Long	AngleFloat32	Longitude degree (-180 to 180)		

Notes Table

Label	Semantics
FuelEconomyBase_1	Consider using a float for this for higher accuracy

Object Class Definition Table

Class	Semantics
Car	A car for the Fuel Economy federation

Interaction Class Definition Table

Class	Semantics
LoadScenario	Interaction to set up the scenario
ScenarioLoaded	This interaction confirms that the scenario has been loaded
ScenarioLoadFailure	To be sent by a federate that has failed to load a scenario for some reason
Start	Interaction to Start the simulation
Stop	Interaction to Stop the simulation

Attribute Definition Table

Class	Attribute	Semantics

Car	Name	Name of the car
	LicensePlateNumber	Number on the license plate.
	FuelLevel	Current fuel level of the car at each point in time
	FuelType	Type of fuel for the car.
	Position	Position of the car.

Parameter Definition Table

Class	Attribute	Semantics
LoadScenario	ScenarioName	Scenario file for the simulation
	InitialFuelAmount	Initial amount of fuel
ScenarioLoaded	FederateName	Name of the federate that succeeded in loading the scenario
ScenarioLoadFailure	FederateName	Name of the federate that failed with loading the scenario
	ErrorMessage	Explanation of failure
Start	TimeScaleFactor	How fast will the simulation run compared to real time. 1.0 = real time. 15.0 = one second of real time corresponds to 15 seconds of scenario time

Appendix C: Description of Federates and File Formats

1. Overview

The federation consists of four federates:

1. Master from which the operator can control the entire federation
2. CarSimC which simulates cars and is written in C++
3. CarSimJ which simulates cars and is written in Java
4. MapViewer which displays the cars on a map

There is a shared scenario file that describes the scenario to be run. There are also car description files for each CarSim that describe car instances and performance parameters.

The simulation models are very simple. The federation management is also very simplified but demonstrates some basic principles.

2. About the Shared Scenario

A shared scenario is specified in a text file in the Scenario directory. There may be several scenarios in the directory. In our sample we provide the SanJose scenario and the Cupertino scenario. A scenario specifies:

- The map to use (in this case just a bitmap)
- The destination and the map to use
- The starting position for driving

All cars will drive in a straight line from the start to the destination. This what a scenario file looks like:

```
// Sample scenario file
Map=spain.jpg
TopLeftLat=44.577193
TopLeftLong=-10.522665
BottomRightLat=34.969554
BottomRightLong=5.207599
StartLat=40.401925
StartLong=-3.293953
StopLat=38.709285
StopLong=-9.136848
// End
```

3. The Master Federate

The Master federate is a command-line application implemented in Java.

The Master publishes the LoadScenario, Start and Stop interactions. It subscribes to the ScenarioLoaded and ScenarioLoadFailure interactions.

When started it joins the federation and then presents the following message:

> Welcome to the Master Federate of the Fuel Economy Federation
> Make sure that your desired federates have joined the federation!

It the presents the following menu

> Select a command
> 1. Load the Redmond scenario
> 2. Load the Cupertino scenario
> 3. Start simulating
> 4. Stop simulating
> Q. Quit the Master Federate

The user can give commands over and over again until he selects Quit. The Load commands will prompt the user for the amount of fuel to be used. It will then result in the Master sending a LoadScenario interaction. Command 3 and 4 will result in a Start and Stop interaction respectively.

When a ScenarioLoaded interaction is received it will print out:

> Scenario Loaded interaction received from <federatename>

When a ScenarioLoadFailure interaction is received it will print out:

> Scenario Load Failure interaction received from <federatename>
> Reason: <Error message>

The configuration for this federate is stored in the file <something>. This includes RTI configuration (CRC host and port), federation name, federate name, federate type and more. Some RTIs may require additional configuration using their own configuration files.

4. The CarSimC Federate and its States

The CarSimC federate is a command line federate implemented in C++ that simulates cars.

The CarSim subscribes to the LoadScenario, Start and Stop interactions. It publishes the Car class with the Name, LicensePlate, FuelLevel, FuelType and Position attributes. It also publishes the ScenarioLoaded and ScenarioLoadFailure interaction.

When started it reads the car description files in its Cars directory. It then joins

the federation, registers the car instances and the presents the following message:

CarSimC has joined. Ready to receive scenario.
Press Q at any time to quit.

When it receives a LoadScenario interaction it will load the scenario, position the cars at the starting position, fill them with the specified amount of fuel, reset the time to zero, send a ScenarioLoaded interaction and then present the following message:

CarSimC has loaded scenario <Scenario name>
The following cars are standing by at the starting point:
<Name of car 1>
…
<Name of car n>

If the scenario file could not be found or if it is incorrect a LoadScenarioFailure interaction will be sent and an error message is printed.

When it receives the Start interaction it will start running the cars along a straight line from the current position (initially the starting position) of each car towards the destination. Cars will stop when they reach the destination. It will also present the following message:

CarSimC is now simulating <n> cars

When it receives the Stop interaction it will stop running the cars and show the message.

CarSimC has stopped simulating.

It is now possible for the master to send a Start message to continue using the current location or to set a new scenario. The following state diagram shows the allowed transitions between the three states NoScenario, ScenarioLoaded and Running.

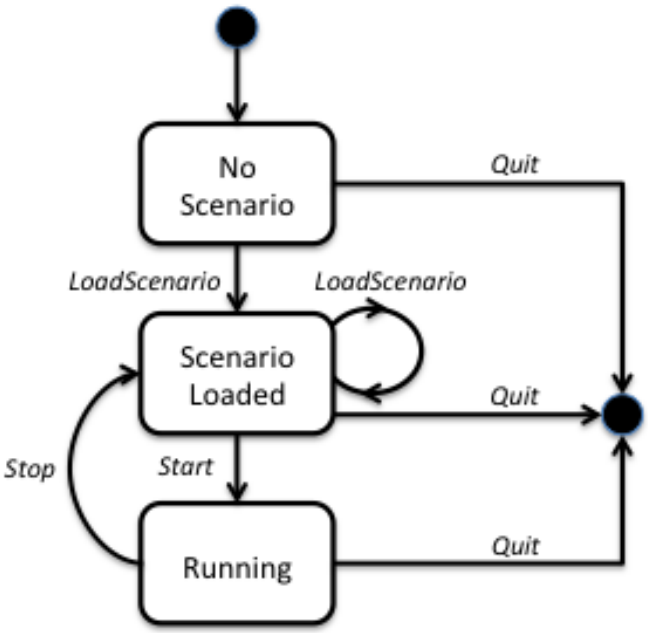


Figure C-1: Internal states in a CarSim federate

As can be understood from this diagram certain interactions from the Master must be rejected in certain states, for example with the messages:

Cannot start when no scenario has been loaded.
Cannot load a new scenario while running
Cannot stop when not running

Let's look at the car description that each federate has in its Cars directory. Each car description file contains the following:

```
// 440d car file
Name=A-Brand 440d
LicensePlate=ABC-123
FuelType=Diesel
NormalSpeed=90
LitresPer100km1=25
LitresPer100km2=15
LitresPer100km3=8
// End
```

The cars will always drive at their "normal" speed, in this case 90 km per hour in this simplified model. The fuel consumption is different during the first minute, the second minute and the third minute and beyond. The high initial fuel consumption is due to the cold start of the engine. A car may run out of fuel which will make it stop.

The configuration for this federate is stored in the file <something>. This includes RTI configuration (CRC host and port), federation name, federate name,

federate type, frame rate and more. Some RTIs may require additional configuration using their own configuration files.

5. The CarSimJ Federate

The CarSimC federate is a command line federate implemented in Java that does exactly the same as the CarSimC federate.

6. The MapViewer Federate

The MapViewer federate is a graphical application implemented in Java that presents a list of the cars to the left and to the right a map with the cars displayed as icons.

The MapViewer subscribes to the LoadScenario, Start and Stop interactions. It also subscribes to the Car class and the Name, LicensePlate, FuelLevel, FuelType and Position attributes.

When receiving s LoadScenario interaction the MapViewer will load the corresponding scenario file, display the specified map and set the simulation time to zero. When a car instance is discovered it will be presented in the list to the left, together with its attribute values. The scenario time is also presented in the display.

12.5. Publish Subscribe Matrix

Interactions:

	Master	**CarSim**	**MapViewer**
LoadScenario	Pub	Sub	Sub
ScenarioLoaded	Sub	Pub	Pub
ScenarioLoadFailure	Sub	Pub	Pub
Start	Pub	Sub	Sub
Stop	Pub	Sub	Sub

Attributes:

	Master	**CarSim**	**MapViewer**
Car.Name	-	Pub	Sub
Car.LicensePlateNumber	-	Pub	Sub
Car.FuelLevel	-	Pub	Sub
Car.FuelType	-	Pub	Sub
Car.Position	-	Pub	Sub

Appendix D: Lab Instructions

The labs do note require any programming. They can be carried out by anyone with a reasonable computer experience.

This chapter provides the following lab instructions

Lab 1: A first test drive of the federation.
Lab 2: Connect, Create and Join.
Lab 3: Developing a FOM for Interactions
Lab 4: Sending and receiving interactions
Lab 5: Developing a FOM for objects
Lab 6: Registering and discovering object instances
Lab 7: Updating objects
Lab 8: Running a distributed federation

1. Installing the HLA Evolved Starter Kit

Before you try any of the labs you need to run the installer for the HLA Evolved Starter Kit.

Advanced: Programmers may choose to modify and extend the code of these federates after completing the above labs. In such case you need to install the suggested development environments, which are:

Platform	Language	Environment
Windows	C++	Visual C++ 10.0 Express or full version
Linux	C++	gcc 4.1 or later
Windows&Linux	Java	IntelliJ Community Edition + JDK 1.6

The installation kit contains the following files that are helpful when modifying the code:

- For C++ there are Visual Studio VC10 project files and Make files (32 bit)
- For Java there are IntelliJ project files and Ant build files.

Note that the installation kit contains a ReadMe file with additional technical information.

2. Support

Community support is available for the HLA Tutorial and the sample federates as follows:

1. Go to www.stackoverflow.com
2. Use the tag HLAstarterkit

There is no commercial support for the HLA Tutorial or the sample federates.

Lab D-1: A first test run of the federation

In the first lab we will install, start up and run the federation.

1. Installation

If you don't already have any Pitch software installed then download and install the following software in the following order:

1. Pitch pRTI Free
2. Pitch Visual OMT Free
3. HLA Evolved Starter Kit

If you already have a commercial version of Pitch pRTI (version 4.4 or higher) installed then don't install Pitch pRTI Free. You should then run the samples using your commercial product. Note that Pitch pRTI Free only allows for two federates at the same time, except when running the Fuel Economy Federation and the Restaurant Federation.

If you already have a commercial version of Pitch Visual OMT (version 2.2 or higher) installed then don't install Pitch Visual OMT Free. You should then run the samples using your commercial product.

2. Starting up the federation

On the Start Menu, locate the following items:

Pitch pRTI Free -> Pitch pRTI Free

Fuel Economy-> Master
Fuel Economy-> CarSimJ
Fuel Economy-> CarSimC
Fuel Economy-> MapViewer

To start the federation:

1. Start Pitch pRTI Free
2. Start all of the above federates in any order. Look in the pRTI Window. You should see federates appearing.
3. In the Master window: select a scenario using the menu
4. In the Master window: give the start command
5. In the MapViewer window, watch the cars move
6. Terminate the applications when you are done

Extensive documentation for Pitch pRTI Free (configuration, troubleshooting, etc) is available in the Start menu at

Pitch pRTI Free -> Documentation -> pRTI Users Guide

Lab D-2: Connect, Create and Join.

In this lab we will study the Connect, Create and Join steps in the federation.

1. About the Pitch pRTI User Interface

The following picture shows the most important part of the Pitch pRTI user interface:

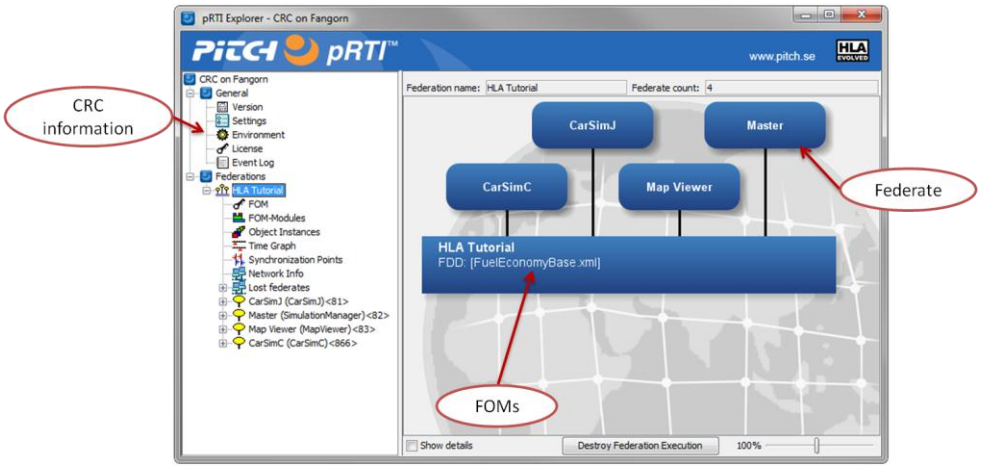


Figure D2-1: The Pitch pRTI user interface

2. Study and verify the start-up

Follow these steps

1. Start the Pitch RTI Free (CRC)
2. Note that there is no federation and no federates visible in the RTI
3. Start the Master federate. Look in the RTI user interface. You should see the Federation Execution named “Fuel Economy” as well as the “Master” federate.
4. Start the other three federates (CarSimC, CarSimJ and MapViewer) and verify that they join the federation as expected.
5. Terminate the federates, one by one, except for the Master, and verify that they disappear from the federation accordingly.
6. Finally terminate the Master federate and verify that it disappears from the federation and that the federation execution goes away.
7. Shut down the RTI.

3. A look at source code and APIs

To study the C++ and Java source code of the fuel economy federates, open the source code directory using the following Start menu item. Note that the source code of the different federates are stored in different subdirectories.

HLA Evolved -> Fuel Economy-> Source code

To study the C++ and Java APIs of HLA Evolved, use the following Start menu items:

Pitch pRTI Free -> Documentation -> HLA Evolved Doxygen
Pitch pRTI Free -> Documentation -> HLA Evolved JavaDoc

Do the following:

1. Open the CarSim Source code (C++ or Java of your choice). Look at the following classes:

   CarSimJ: se.pitch.hlatutorial.carsim.hlamodule.HLAinterfaceImpl
   CarSimC: /source/HLAmodule.cpp

   Locate the code that performs the Connect, Create Federation and Join Federation Execution calls. Note the exception handling which, for clarity, has not been included in the pseudo code in the main text of this tutorial.

2. Locate and study these three services in the HLA Evolved APIs (Doxygen and JavaDoc above).

3. If you have the IEEE 1516-2010.1 specification available, read more about these services in the standard.

4. Perform step 1-3 for the three services Resign Federation Execution, Destroy Federation Execution and Disconnect.

Lab D-3: Developing a FOM for Interactions

In this lab we will study the FOM and the interactions in particular.

1. About the Pitch Visual OMT User Interface

The following picture shows the most important part of the Pitch Visual OMT user interface:

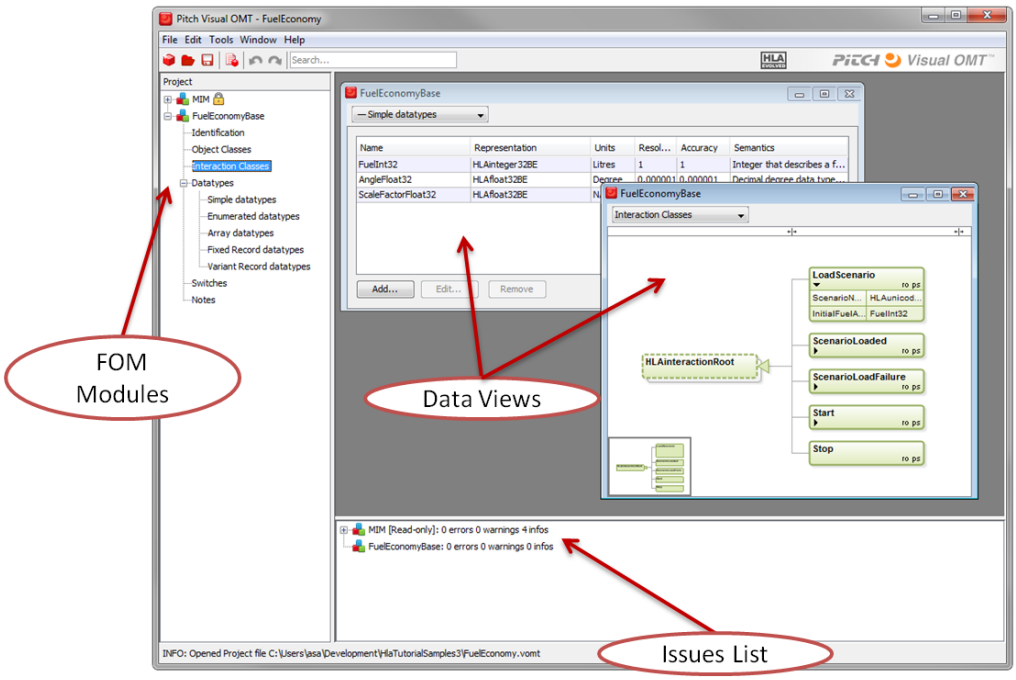


Figure D3-1: The Pitch Visual OMT user interface

In the left pane you can open and close different FOM modules in the project. By double-clicking an item in the left pane, for example Identification Table, you will open this table.

2. Locating the FOM

To open the FOM start Pitch Visual OMT Free and select the Fuel Economy FOM project in the start dialog.

3. A look at interactions

Do the following:

1. Open the FOM and locate the Fuel Economy FOM module
2. Inspect the contents of the Identification table
3. Inspect the tree of interaction classes. Open and close their parameter list

using the small triangle. Double-click on the LoadScenario interaction to inspect it.

4. Go to the data types and look at the simple data types. Here you should inspect the FuelInt32.

5. Now go to the MIM FOM module. This module contains a lot of things that are predefined in the HLA standard.

6. Inspect that various data types in the MIM. Note that all the predefined items are prefixed with HLA.

7. Now try and modify some parts of the FOM. Add your own interaction Refuel with a parameter MaxMoneyAmount. Add a data type EuroType32.

8. Note that you cannot save your modified FOM using the Free version of Pitch Visual OMT.

9. Shut down Pitch Visual OMT

Lab D-4: Sending and receiving interactions

In this lab we will study the services used for sending and receiving interactions.

1. Study and verify declarations

Follow these steps:

1. Start the RTI (CRC)

2. Start the Master federate. Look at the pRTI user interface. Select the Master federate and look at the Declarations

3. You should see that the Start, Stop and LoadScenario interactions are Published. The ScenarioLoaded and ScenarioLoadFailure interactions should be Subscribed.

4. Start the CarSimC federate. Look at the pRTI user interface. Select the CarSimJ federate and look at the Declarations

5. You should see that the ScenarioLoaded and ScenarioLoadFailure interactions are Published. The Start, Stop and LoadScenario interactions should be subscribed

2. Study and verify send and receive services

Follow these steps:

1. In the Master federate, give a LoadScenario command which initiates a LoadScenario interaction.

2. Look at the CarSimC federate and see that it prints a message that it received the interaction.

3. When the CarSimC federate has loaded the scenario it will send a ScenarioLoaded interaction. Look at the Master federate and verify that it prints a message that it has received the interaction.

4. Advance users may switch on the trace for a federate in the RTI user interface.

5. Shut down the federation

3. A look at source code and APIs

Do the following:

5. Open the CarSim Source code (C++ or Java of your choice). Look at the following classes:

   CarSimJ: se.pitch.hlatutorial.carsim.hlamodule.HLAinterfaceImpl
   CarSimC: /source/HLAmodule.cpp

   Locate the code that performs the Get Interaction Class Handle and Send Interaction calls and that implements the Receive Interaction callback. Note the exception handling which, for clarity, has not been included in the pseudo code in the main text of this tutorial.

6. Locate and study these three services in the HLA Evolved APIs (Doxygen and JavaDoc above).

If you have the IEEE 1516-2010.1 specification available, read more about these services in the standard.

Lab D-5: Developing a FOM for object classes

In this lab we will study the FOM and the object classes in particular.

1. A look at object classes

Do the following:

1. To open the FOM start Pitch Visual OMT Free and select the Fuel Economy FOM project in the start dialog.

2. Inspect the tree of object classes. There are actually only two classes: the Object Root and the Car. Open and close the attribute list of the Car class using the small triangle. You may want to increase the width of the column to read the full names. Double-click on the Car class to inspect it.

3. Go to the data types and look at the Enumerated data types. Here you should inspect the FuelTypeEnum32.

4. Go to the data types and look at the Fixed Record data types. Here you should inspect the PositionRec. Note how it builds upon the AngleFloat32 data type. Can you find the definition of that data type?

5. Now try and modify some parts of the FOM. Add your own class FuelStation with attributes Name, Position, HasDiesel and HasPetrol. Add more data types.

6. Note that you cannot save your modified FOM using the Free version of Pitch Visual OMT.

7. Shut down Pitch Visual OMT

Lab D-6: Registering and discovering object instances

In this lab we will study the services used for registering and discovering object instances.

1. Study and verify declarations

Follow these steps:

1. Start the RTI (CRC)

2. Start the Master federate. Then start the CarSimC and the MapViewer federate. Look at the pRTI user interface. Select the CarSimC federate and look at the Declarations

3. You should see that the Car object class (including all of the attributes) is Published.

4. Start the MapViewer federate. Look at the pRTI user interface. Select the MapViewer federate and look at the Declarations

5. You should see that the Car object class subscribed

2. Study and verify registration and discovery services

Follow these steps:

1. In the Master federate, give a LoadScenarion command.

2. Look in the pRTI GUI at the Fuel Economy federation. Look at the Object Instances of the federation. You should now see the Car instances that have been registered in the federation

3. Look in the pRTI GUI at the CarSimC federate. Look at Known Instances and verify that the cars are known (actually created) by this federate.

4. Look in the pRTI GUI at the MapViewer federate. Look at Known Instances and verify that the cars are known (actually discovered) by this federate.

5. Shut down the federation

3. A look at source code and APIs

Do the following:

1. Open the CarSim Source code (C++ or Java of your choice). Look at the

following classes:

CarSimJ: se.pitch.hlatutorial.carsim.hlamodule.HLAinterfaceImpl
CarSimC: /source/HLAmodule.cpp

Locate the code that performs the Register Object Instance call and that implements the Discover Object Instance callback. Note the exception handling which, for clarity, has not been included in the pseudo code in the main text of this tutorial.

2. Locate and study these three services in the HLA Evolved APIs (Doxygen and JavaDoc above).

3. If you have the IEEE 1516-2010.1 specification available, read more about these services in the standard.

Lab D-7: Updating objects

In this lab we will study the services used for sending updates for attributes of objects as well as for reflecting these attributes in subscribing federates.

1. Study and verify declarations

In the first lab you have learned how to start and run the federation. As part of this lab you could see cars in the MapViewer (subscriber) that were simulated by the CarSims (publishers).

Do the following:

1. Start up the RTI and all federates. Load a scenario and start simulating.

2. Verify that you can see the cars moving in the MapViewer.

3. In the pRTI user interface: select the CarSimC federate and switch on tracing of RTI calls and callbacks. You may consider switching off the tracing after a few seconds to limit the number of trace printouts.

4. Look at the CarSimC window. Study the calls that the federate makes to Update Attribute Values service.

5. In the pRTI user interface: select the MapViewer federate and switch on tracing of RTI calls and callbacks. You may consider switching off the tracing after a few seconds to limit the number of trace printouts.

6. Look at the MapViewer window. Study the Reflect Attribute Value callbacks that the RTI makes to the federate.

7. Terminate the federation.

2. A look at source code and APIs

Do the following:

1. Open the CarSim Source code (C++ or Java of your choice). Look at the following classes:

   CarSimJ: se.pitch.hlatutorial.carsim.hlamodule.HLAinterfaceImpl
   CarSimC: /source/HLAmodule.cpp

   Locate the code that performs the Update Attribute Values calls and the Reflect Attribute Value callbacks. Note the exception handling which, for clarity, has not been included in the pseudo code in the main text of this tutorial.

2. Locate and study these three services in the HLA Evolved APIs (Doxygen and JavaDoc above).

3. If you have the IEEE 1516-2010.1 specification available, read more about these services in the standard.

Lab D-8: Running a Distributed Federation

The following is an advanced exercise for anyone interested in trying out a distributed federation. It assumes that you have at least two computers on a network.

1. Running with two computers

We will call the Computer 1 and Computer 2. For simplicity you may need to disable the Windows/Linux firewall on your computers.

We will distribute the applications as follows:

Computer 1: Central RTI Component, Master, CarSimC
Computer 2: CarSimJ, MapViewer

1. Make sure that Pitch pRTI Free and the HLA Evolved Starter Kit are installed on both computers.

2. We will use Computer 1 for the Central RTI Component. Start up the Pitch pRTI Free on this computer. Remember the IP address of this computer, for example 192.168.1.23. This address can be located using the “ipconfig” (Windows) or “ifconfig” (Linux) command.

3. Start up the Master and the CarSimC on Computer 1 and verify that they join the RTI.

4. Modify the following files on Computer 2:
    a. CarSimJ config
    b. MapViewer config

5. In both of the above files modify the CRChost value to the IP address for computer 1. The resulting line may be for example
    a. CRChost=192.168.1.23

6. Start the CarSimJ and MapViewer federates on Computer 1.

7. Check in the Pitch pRTI Free user interface that all four federates have joined.

8. Start the simulation using the Master federate.

2. Running with even more computers

You may move all federates to different computers. The only requirement is that you have the RTI installed since each federate needs to use the RTI libraries in that installation. You also need to modify the configuration file for each federate

so that it connects to the CRC.

3. Using the Web View for iPad/Android/iPhone

Read the Pitch pRTI Users Guide on how to start the Web View for Pitch pRTI The Web View enables you to connect to the RTI and manage federations using regular web browsers or tablets (iPad/Android) and even mobile phones.

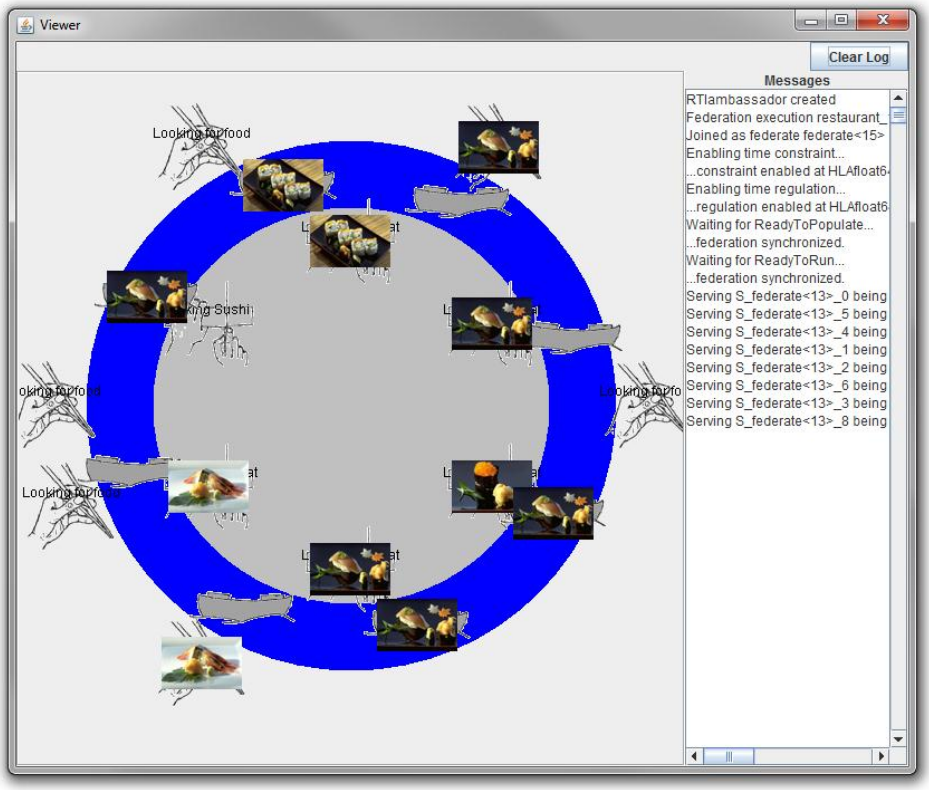
Appendix E: The Sushi Restaurant federation

When HLA was first introduced one of the more advanced samples was the Restaurant federation. It uses a large number of HLA services. This federation, originally developed for HLA 1.3, has been migrated to the most recent version of HLA. A full description of the federation is provided in the book “Creating Computer Simulation Systems - An Introduction to the High Level Architecture” by Fred Kuhl, Judith Dahmann and Richard Weatherly, ISBN: 9780130225115.

Note that this federation was originally developed using older versions of both Java and HLA. This means that some of the code may not use what we today consider best practice. Nevertheless, the rich set of HLA features used still makes it an interesting federation.

1. Overview of the federation

The federation models a classic sushi restaurant where the chefs prepare sushi and places them on boats that transport them to the customers.



Picture E-1: Sushi Federation Viewer

2. How to run the federation

This federation is installed as part of the HLA Evolved starter kit. See Appendix D-1 for details.

On the Start Menu, locate the following items:

Start -> HLA Evolved -> Sushi Restaurant -> Production
Start -> HLA Evolved -> Sushi Restaurant -> Transportation
Start -> HLA Evolved -> Sushi Restaurant -> Consumption
Start -> HLA Evolved -> Sushi Restaurant -> Viewer
Start -> HLA Evolved -> Sushi Restaurant -> Manager

To run the federation, start pRTI Free and then the above federates. Note that the Manager federate should be the last one that you start.

The source code is available in a directory that is opened using:

Start -> HLA Evolved -> Restaurant -> Source Code

To inspect the FOM, start Pitch Visual OMT Free and select the Sushi Restaurant project.

3. Production Federate

This federate manages a collection of Chef object that produces sushi objects. When a boat passes close to the chef the chef can give the sushi away and place it on a boat.

4. Transportation Federate

The transportation manages a collection of boat object. When a boat is close to a chef that has produced a sushi the production federate can offer the transportation federate to take ownership of the sushi. The sushi will then be loaded on the boat and transported to potential costumers

5. Consumption Federate

The Consumption federate models a number of consumers. When a boat containing a sushi passes close to a customer it can take over ownership of the sushi and eat it.

6. Manager Federate

The manager federate keeps track of the federates in the federation and is responsible for settings synchronization points (ex ReadyToRun). It is also paces the time to keep the federation advancing at the desired speed.

7. Viewer Federate

The viewer federate displays the chefs, boats, customers and sushi in a graphical view. The federation simulates the activities at a restaurant.

Appendix F: A summary of the HLA Rules

HLA provides ten rules for federates and federation in the standards document "HLA Rules". Here is a simplified version of these rules.

The rules use the concept of a Simulation Object Model, SOM. The SOM is very similar to a FOM. It is based on the same format, the HLA object model template. It describes what information one particular federate can publish and subscribe. The following example illustrates the difference

FOM	Car.Position	Publish/Subscribe
SOM for CarSim	Car.Position	Publish
SOM for MapViewer	Car.Position	Subscribe

Note that the FOM relates to one particular federation. A SOM relates to what a federate can publish and subscribe in any potential federation. You may think of it as a brochure advertising the capabilities of a federate. In a particular federation the federate may only publish and subscribe to a subset of the object and interaction classes in the SOM.

1. Federation rules

1. All federations shall have a FOM
2. The federates shall store the attribute values, not the RTI
3. For the information that is described in the FOM, all data exchange shall be performed using the RTI
4. Federates shall only interact with the RTI using the services described in the interface specification
5. Each attribute of any object instance may only have one federate that owns is (and is thus allowed to update it) at any given time

2. Federate rules

6. Federates shall have a SOM
7. Federates shall send and receive data according to their SOM
8. Federates shall be able to transfer ownership according to their SOM
9. Federates shall follow their FOM with regards to how they send/receive attribute updates
10. Federates shall manage their internal time in such a way that it can be coordinated with the data exchange with other federates using HLA services

Getting started with FOM Modules

Björn Möller
Björn Löfstrand

bjorn.moller@pitch.se
bjorn.lofstrand@pitch.se

Keywords:

HLA Evolved, FEDEP, FOM Modules

ABSTRACT: *One of the most important new features in HLA Evolved is the ability to provide the Federation Object Model (FOM) in the form of several modules. While the standard provides a complete and stringent specification of these concepts, the purpose of this paper is to provide a user-friendly introduction as well as some guidelines for their practical use. A small example FOM is used in the paper to illustrate the principles.*

First of all it is important to understand where FOM modules fit into the FEDEP process and how it further enhances the process of building interoperable and reusable systems. Some general comments about best-practice for FOM modeling are also given.

The following aspects are then covered

- Understanding the predefined HLAstandardMIM module

- Use of standalone versus dependent modules

- Describing different aspects of a domain in different FOM modules

- Developing more general modules that can be reused across federations

- Handling of commonly used data types

- Extending existing modules with subclasses, including guidelines for using "scaffolding" classes

- Strategies for the required Switches table with suggested default values

- Providing FOM modules at Create versus Join time

Finally a list of additional ideas and aspects, that are not covered in the paper, is given for the reader to investigate on his own.

1. Introduction

The High-Level Architecture (HLA) was first developed in the mid 90's and standardized as HLA 1.3 [1] in 1998. It has since gone through two major revisions: IEEE 1516-2000 [2] and, soon to be released, IEEE 1516-2009 [3][4] a.k.a. HLA Evolved. The latest revision contains a large number of new features while maintaining all of the previously available functionality. Some examples of new features are fault tolerance support [5], Web Services support [6] and FOM modules [7]. A previous paper [8] lists all major and most minor technical improvements. Most of the new technical features relate to the HLA Interface Specification and thus the runtime behavior of HLA. FOM Modules, on the other hand, primarily relates to the HLA Object Model Template specification and, more importantly, to how to create information models for interoperability. This paper is dedicated to FOM modules, how they can be used and their relation to the overall process of building interoperable systems.

1.1 About this paper

This is an introductory paper, not a specification. It shows, step-by-step, how to use FOM modules on a basic to intermediate level. Some familiarity with HLA in general and FOM development is highly recommended. We have chosen not to give the full theory and specification upfront but to cover typical FOM module design. At the end of this paper we mention some more advanced topics. The full and

exact specification can be found in the HLA Evolved standard.

1.2 About design

The reader of this paper is most likely interested in getting an introduction to FOM modules with clear and unambiguous design rules. In practice, most of FOM module design, like any engineering, is about striking a balance. When we create a reusable information model, exactly how general and reusable should it be? To what degree shall it be easily adaptable to current simulation systems with all of their quirks and peculiarities? Can we predict all possible future uses of this information model? Shall we build a quick-and-dirty solution or an extremely clean model? To what extent shall we reuse and modify existing models?

There are no final answers to these questions. There are however certain factors that should be taken into consideration, for example:

- What is the expected life-span of the federation and the FOM?
- What user communities are involved, now and in the future? What are their current and expected requirements?
- What budget is available?
- Which subject matter experts are available?
- What is the planned reuse of the FOM?
- What existing efforts can we build upon?

2. FOMs – an Overview

When two or more systems are to interoperate they need to exchange information using a data model, sometimes called an Information Exchange Data Model (IEDM). In HLA this model is called the Federation Object Model (FOM). The FOM describes information that is to be shared between different federates. A FOM contains the following:

- An identification table describing things like the purpose, origin and author of the FOM
- Shared object classes with associated attributes.
- Shared interaction classes with associated parameters.
- Data type definitions for attributes and parameters
- Dimensions used for Data Distribution Management (DDM) filtering
- Time representation used
- Synchronization points used
- Data types for user supplied tags
- Transportation types, in most cases only Reliable and Best Effort
- Notes, typically annotations made during the FOM development
- Runtime switches for the RTI

We will now, step by step, look at some important properties of the FOM

The FOM is based on what other systems need to consume. A common misconception among inexperienced federation builders is that the developer of each system can decide what information they want to send to others. In practice the FOM must be based on what information other systems need to consume from your system. In most cases this information needs to be described in a more generalized form than the internal representation of each system.

The FOM is usually a simplification compared to the internal domain representation of each system. Each system will typically contain a rich internal data model of the reality it seeks to imitate. The information exchange data model focuses on what other systems need. There is no need to expose all the internal details required to produce a high fidelity model of an entity or a process. There are of course several exceptions to this statement. A data logger that has very limited internal representation is an example of such an exception.

Example: a simulation contains an aircraft model which includes a very detailed aerodynamics model. The federation where this simulation is used only looks at the position, speed and type of aircraft. Even if the simulation developer has a strong urge to expose intricate aerodynamics attributes they still should not go into the FOM.

We will now look at how the FOM is key to achieving interoperability and reuse of our simulations.

Using an information bus, such as HLA, with a shared information model, such as the FOM, enables composability. In some interoperability standards, such as web services, the focus is to connect exactly two systems using an information exchange data model that is specific to this pair of systems. If instead an information bus with a shared information exchange data model is used, it is possible to achieve composability. The producers and consumers of data do not need to have detailed knowledge about each other, only about the information being exchanged. Different sets of systems can be combined at different occasions for different purposes. New systems can replace older systems without affecting other systems. The total set of systems can easily grow over time.

A carefully designed FOM facilitates long-term reuse of simulations. If a FOM is designed to meet

the needs of a larger set of federations this means that it will be easier to reuse one system from one federation in another federation. The simulation is already adapted to the FOM. Reusing the FOM means that it is easier to reuse the systems. Note however that it is not enough. For each federation you still need to evaluate each federate, for example if it contains the right fidelity and if it produces and consumes the right information.

The FOM is part of a federation agreement. The federation agreement documents things like who is supposed to produce and consume information, when the data is provided, frequency, accuracy, etc. It also describes how the execution is started and stopped and how the scenario is loaded. It describes expected responses to various events from different federates, and many other things. The FOM gives a good description of the types and format of data that is exchanged but it may prove useless if you don't have access to the full federation agreement. Interoperability doesn't happen by chance and all federations build upon some sort of agreement between federate developers.

3. Some basic best practices for FOMs

There are a lot of best-practices for developing Information Exchange Data Models in general and FOMs in particular. Unfortunately a lot of this is scattered in literature and papers.

As a general advice it is wise to build upon generally accepted information models within your domain, in particular on the attribute or parameter level. There is no need to invent a new coordinate system if most of the participating systems already use lat/long or geocentric coordinates. In many cases there already exists enumerations, for example country codes. Some additional rules of thumb that are useful when developing FOMs are:

Do not use runtime identifiers, such as RTI handles, in the FOM. Imagine the case where one object, like a pilot, references another object, like an aircraft. A first approach may be to use the HLA object handle of the aircraft for this reference. If this data is logged and examined, for example for after action review purposes, this reference will be impossible to resolve and the relationship is lost. Using the "marking" attribute of the aircraft or another domain-oriented name or label is a better choice. It is also important that the federation agreements specifies how this reference is guaranteed to be a unique identifier.

Use records wisely. If several attributes are strongly related and an update of one attribute doesn't really make sense without updating another attribute, then put both of them in a record. A good example is a position that is described using latitude and longitude. If you, on the other hand, put more or less all of the information about an object in a record you will waste bandwidth and loose scalability. In addition to this you will prevent other federates from being able to subscribe only to the information that they really need.

Do not subclass without a good reason. It is easy to confuse the object class hierarchy in the FOM with object oriented languages where different subclasses exhibit different behavior. A better comparison for a FOM would be a corporate data base where data is stored and fetched by different applications. There are really only two main reasons for creating a subclass. The first is interest management where federates want to subscribe to one particular class (for example aircraft) but not another (for example submarine) given a common parent class (like vehicle). The second is that there is a requirement to add one or more attributes or parameters that is relevant to one class but not another.

4. A FEDEP perspective for FOM Module developers

Before developing a FOM it is important to understand the Federation Development and Execution Process (FEDEP) [9]. It provides a framework for developing federations. Federations can be built both from a combination of existing simulation systems, more or less adapted, and new systems. This makes the process slightly different from other processes which assume that everything is built from scratch.

The FEDEP isn't very detailed on a technical level. This is intentional since it is intended to harmonize with, not replace, your current development methodology and software engineering processes.

It is very important to understand the three first steps in FEDEP. They provide key information for your FOM module development. Later steps may provide additional information that can be used to further refine your FOM modules. Figure 1 shows the FEDEP process with some FOM module consideration are:

In step one, *Define Federation Objectives*, the overall goals and constraints of the federation is decided. What is our ambition with the federation? Is it a one-time event or in support of a full-fledge training environment? The objectives of the federation sets the stage for the basic design of the FOM. The level of flexibility, reusability and dynamics in terms of information exchange can often be determined from these initial objectives and metrics. It is important to understand how overall design aspects like reuse affects the way we design a FOM. Are we going to develop a federation that will have a short or long life span? How is this effort related to other efforts and what type of reuse is expected. Is there a requirement, a budget and subject matter experts available for creating more generally reus-

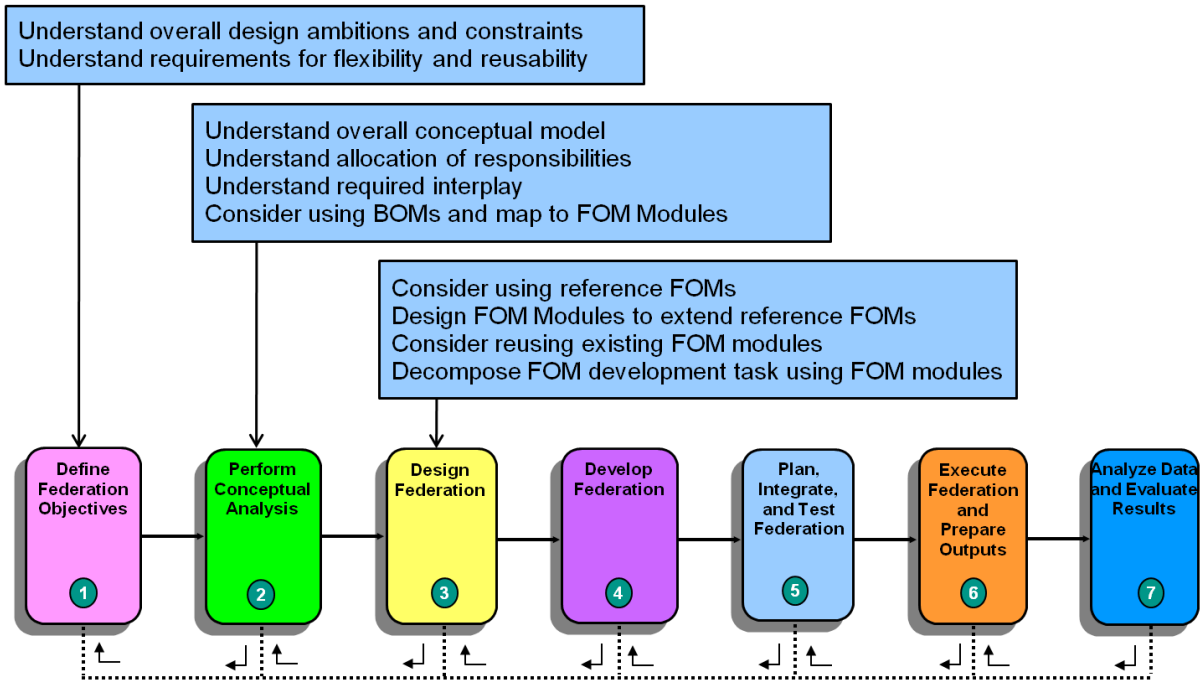


Figure 1: The FEDEP Process and FOM Modules

able FOM modules or should the effort only focus on short-term goals?

In step two, *Perform Conceptual Analysis*, the conceptual model of the simulation is developed based on one or more typical scenarios. Through the conceptual model all relations and interplay between simulated entities must be defined. When distributing the responsibility of modeling all that is defined in the conceptual model these relationships must be upheld. Any interplay between entities modeled in different federates must be manifested as information exchange between federates using constructs defined in the FOM. It is also important to note that the way relationships and interplay is defined in the conceptual model may be captured differently in the FOM depending on how the modeling responsibility is distributed among federates. In this step existing conceptual models and mappings to FOM constructs can be used. A good reuse element in this step are Base Object Models (BOM) [10] [11] that maps conceptual elements to FOM constructs.

In step three, *Design Federation*, the more technical FOM module development work begins. Simulators that are to become federates are selected. The required functionality and modeling responsibility are allocated to federates. It is now time to examine what information that each federate needs to consume from other federates to carry out its work. Information exchange requirements can be derived from the conceptual model, supporting BOMs and the federation specific allocation of modeling responsibilities. The design of the federation must be made carefully to meet all federation requirements. The FOM is defined to support the necessary information exchange between federates.

The use of reference FOMs promote reuse and, to some level, interoperability. When designing a federation it is important to carefully analyze the pros and cons of using an existing FOM. The trade-off between a FOM optimized for a specific federation and the use of a more standard FOM is a key decision point in the development of a federation. Details in the conceptual model specific to the federation are likely not to supported in a standard FOM and therefore require some FOM modifications or a completely different FOM. Although these customized FOMs increases interoperability they also reduce the opportunity of reusing existing federate based on the standard FOM.

In many organizations more and more internal FOM modules are expected to exist. These may be evaluated for potential reuse.

For the development of larger FOMs the task can be decomposed into different aspects of the FOM. Different teams can now take responsibility for the development of different FOM modules. Common concepts, data types in particular, may be shared in a common module.

During the rest of the FEDEP process the need for minor adjustments may arise. However, the major driver for FOM updates will be new requirements for the entire federation, for example if new types of objects are to be modeled or if new sequences of events are to be modeled.

5. General about HLA Evolved FOM Modules

In earlier versions of HLA the FOM was provided as one monolithic FOM. If you look at many real-life FOMs, for example the RPR FOM, you will notice that they are big and contain information in many different areas. HLA Evolved offers the opportunity to provide FOM data as modules. Still, the sum of all FOM modules must result in a valid FOM.

5.1 What to use FOM modules for

A long time ago people would develop software programs as one large piece of code. Time has changed and nowadays we develop in a more modular way, producing libraries, modules, classes and similar building blocks. The main principle for how a solution is divided into modules is usually called "separation of concern" where each building block should have, in some sense, a clear and separate purpose and responsibility. There are many similarities between code modules and FOM modules but also some slight differences. The FOM is mainly a contract between different systems. It describes important aspects of how these systems are to interoperate while the responsibility for carrying out any particular work lies within the systems. One important benefit of most types of modularity is that a solution can be developed, provided and reused in a modular way.

FOM modules can be used for several purposes, for example:

- You can have different working groups develop different parts of a FOM in a more convenient way. You may for example have a radio specialist group develop the "Radio FOM module" while the aircraft specialists develop the "Aircraft FOM module".
- You can put extensions to a reference FOM in a separate FOM module. This will prevent you from getting modified, "non-standard" reference FOMs. More importantly, when you merge federates or federations that use extended reference FOMs, it is easy to inspect what extensions that have been made and possibly to merge them.
- You can achieve extended reuse of some aspects of a FOM. If you want to promote a standardized way to start and stop all of your federations, irrespective of domain, you may put these interactions in a separate FOM module.
- You may also add more FOM modules to an already running federation, thus extending the scope of the FOM.

5.2 Content of FOM modules

Note that a FOM module may contain a subset of all tables, for example only data types. An identification table, describing things like the author and the purpose of the module, is always required for documentation purposes. This table will not be used when several FOM modules are combined.

Unless you are an experienced FOM developer your first FOM modules will typically contain the following HLA Object Model Template (OMT) data:

- An identification table
- Some shared object classes with attributes
- Some shared interaction classes with parameters
- Some data types for attributes and parameters
- Possibly a time representation table if your federation is time managed
- A switches table with runtime switches, as described later.

5.3 Using FOM Modules at runtime

There are two RTI services when you can provide FOM modules: the *Create Federation Execution* call and the *Join Federation Execution* call. One important improvement here is that you can indeed introduce new object and interaction classes into an already running federation. This, obviously, assumes that there is more than one federate that use these new classes.

How and when FOM modules are provided should be documented in your federation agreement. The two most obvious strategies are:

Strategy 1: Each federate supplies all FOM modules that they require upon Join. This is the preferred strategy for flexible federations where you want to be independent of join order. It supports early and late joiners and thus federates rejoining after a fault. The downside is that you need to make sure that all federates have access to the most recent version of the FOM modules that it required.

Strategy 2: One master federate creates and joins before any other federate. It loads all required FOM modules. This requires a highly coordinated startup but allows for a more strict centralized control.

You may also use strategy 2 for commonly used FOM modules and add more specialized FOM modules later.

6. A First Look at a Tiny FOM Module

First of all we will look at a really small FOM module. The module is illustrated in figure 2. It contains only three things:

- An object class called Train.
- An attribute Speed for the train
- A data type, SpeedInt64 for the speed.

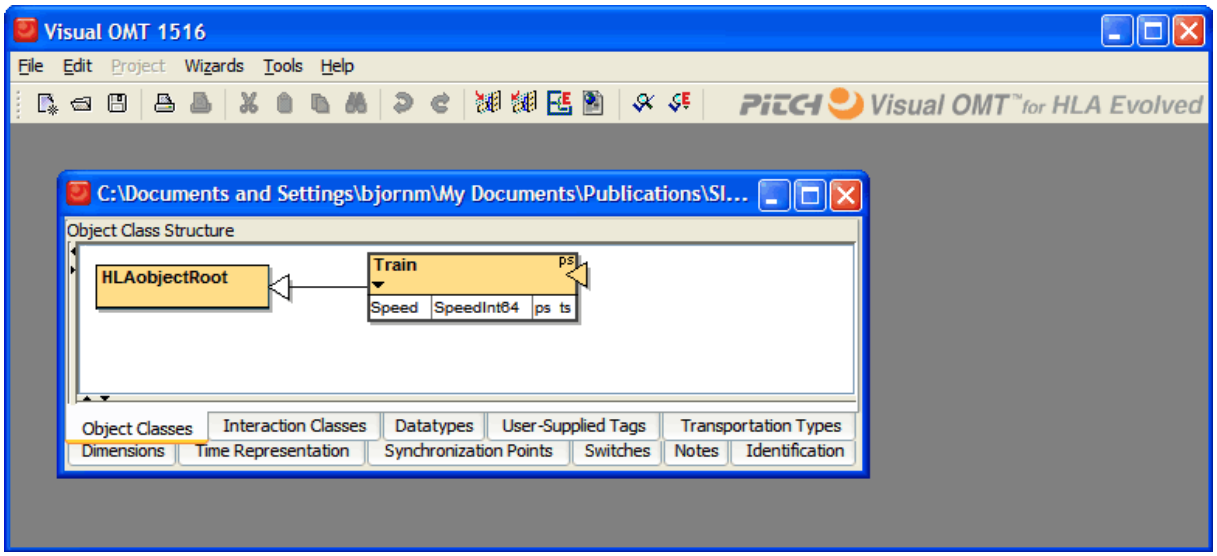


Figure 2: A Tiny FOM Module

In addition to this is contains an identification table. The FOM is shown in XML format in appendix A.

The most interesting thing here is what is not shown in the figure. The Manager.Federate and Manager.Federation class from the MOM isn't included. In fact, all of the MOM and the predefined HLA classes are excluded since they are available in a separate, pre-defined module called HLAstandardMIM, as described later. The class HLAobjectRoot doesn't have any attributes, not even the predefined HLAprivilegeToDeleteObject. The definition of HLAobjectRoot is an empty placeholder, called a *scaffolding* class definition. The purpose is simply to indicate where in the class hierarchy a new class shall be inserted. More about this later.

7. An example of Several FOM modules

We will now look at a sample federation to understand how FOM modules can be used. This is not a real life federation but it is partly inspired by real FOMs such as the RPR FOM [12] and the Dutch Railroad FOM [13]. The purpose of the federation is to simulate a national railway system. It consists of the following federates, as shown in figure 3:

TracksAndSignals provides a representation of the track topology and signals and their state across the nation. It loads the topology from a database.

PassengerTraffic simulates trains that carry passengers.

CargoTraffic simulates cargo trains

SafetyTypeA simulates an older train safety system that is in used in some older parts of the railroad system,

SafetyTypeB simulates a similar but more modern safety system.

TTC simulates the Train Traffic Control center operations.

RailVis is a visualization system that displays the state of the simulation.

Master is a federation execution manager that se-

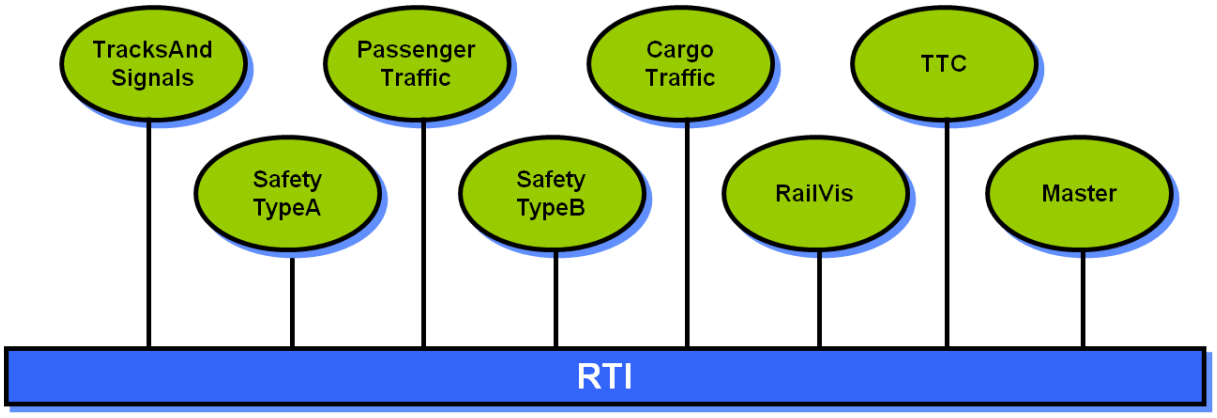


Figure 3: The Sample Railroad Federation

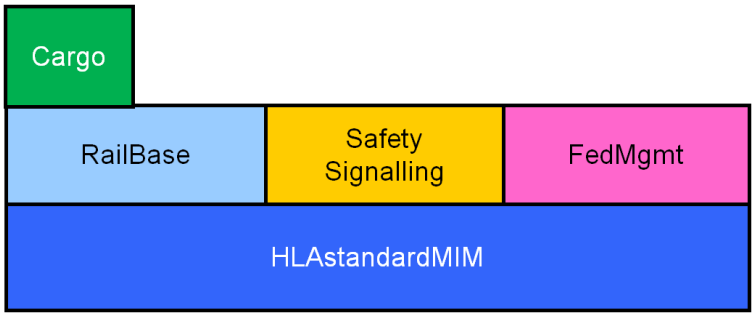


Figure 4: FOM Modules for the Example Federation

lects scenarios, sets the time and starts and stops the federation.

The following FOM modules, as shown in figure 3, are used for this federation:

RailBase contains the basic shared objects that are relevant for all railway simulation such as track segments, signal state, platforms and vehicles such as locomotives and wagons. It also contains the Switches table that the RTI needs as part of the Create Federation Execution call.

Cargo contains a number of specific additions for more advanced cargo transports, in particular heavy or hazardous transports.

SafetySignalling contains a number of more specialized signals that are sent between safety systems, signals and the TTC.

FedMgmt contains interactions that all systems have to obey such as setting scenario time, starting and stopping.

The HLA standard contains a lot of predefined concepts like data types (strings, integers, etc) as well as the roots for the two class hierarchies: Objects and Interactions. These concepts and more are stored in a separate FOM module so we will also need to add the following FOM module to our list:

HLAstandardMIM that contains predefined HLA concepts from the standard. We will build upon this module and use concepts like data types from this module, but we will not modify it in any way.

6.1 Working with object and interaction classes

The RailBase module contains some important classes like TrackSegment, Signal and Train. These three classes build upon the HLAObjectRoot class that resides in the HLAStandardMIM module.

In the Cargo FOM module we want to extend the Train object with the CargoTrain class. This class has a number of advanced properties for example for hazardous trains. The best way to do this is to provide a **scaffolding** (empty) definition for Train with no attributes or other information. We then create the subclass CargoTrain with all of its attributes. See figure 5.

The Cargo module is now said to be **dependent** on the RailBase FOM module. The use of a scaffolding definition has two advantages:

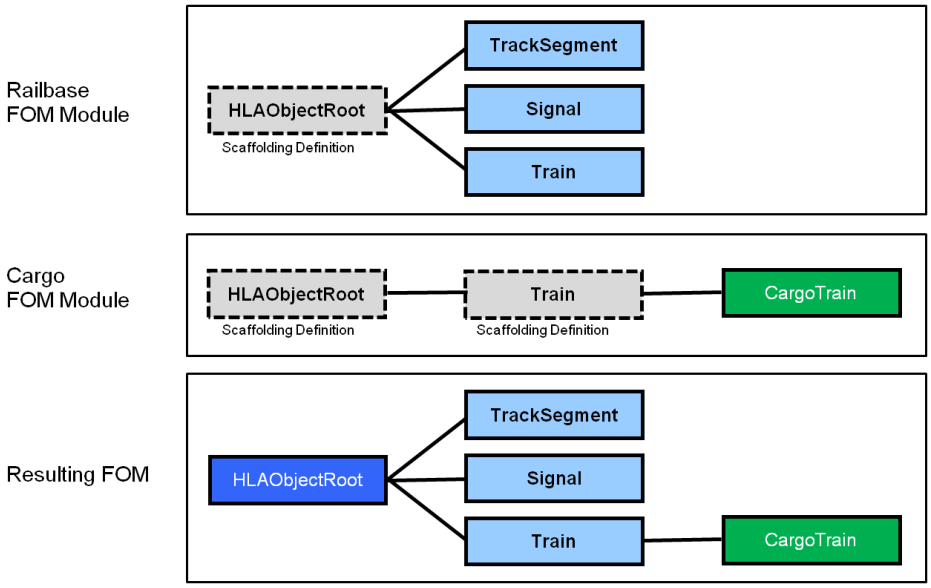


Figure 5: Object Classes in FOM Modules

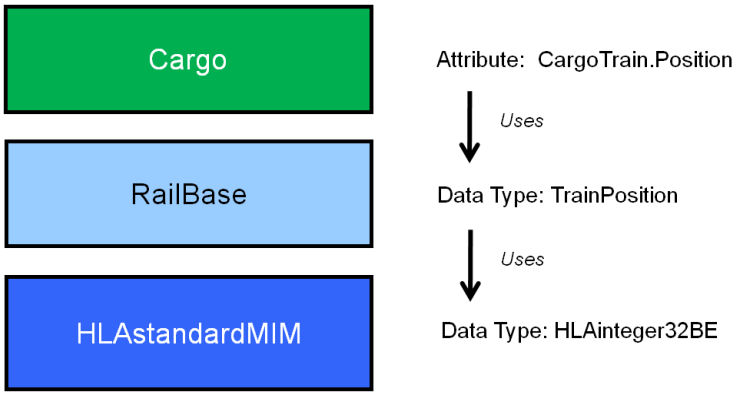


Figure 6: Data Types Dependencies between FOM Modules

1. There is no risk that we will end up with two conflicting definitions of Train
2. The team that is responsible for maintaining the RailBase module can keep refining it without updating the Cargo module.

The RailBase FOM module is a **standalone** module. While it does indeed reference predefined HLA concepts such as the HLAobjectRoot (using a scaffolding definition) it doesn't depend on any user-defined FOM module.

Consider the FedMgmt FOMmodule. This is a module that we may want to reuse across federations. In this case we may want to make it standalone or to have very few dependencies upon other modules.

7.2 Working with data types

No matter if you are simulating trains, signaling systems or train traffic control you will need to reference train IDs, track segments, positions along track segments and lat/long positions. In this example we have decided to put many of these common definitions in the RailBase FOM module. Since the module SafetySignalling uses these data types it will be dependent on the RailBase FOM module. Note that Cargo is dependent on RailBase because of both the Object Classes and the Data Types, as shown in figure 6.

Since we want to make the FedMgmt module standalone we should either avoid using data types from the TrainBase module or we should copy any definitions used into FedMgmt.

In some cases you may also want to put your data types into a completely separate module.

6.3 An introduction to combining FOM modules

The table in figure 7 contains a simplified summary of how FOM modules are combined. It describes data from two hypothetical FOM modules and what the result from merging these would be. Only a subset of the FOM data is used in this example. Note that not all tables need to be present in all FOM modules. Also note that there are some additional constraints that are described later in this paper or in the HLA Evolved specification.

In general the union of the data in all loaded FOM modules is produced. For a few cases the data is required to be equal. This means that the result is independent of load order, assuming that there are no conflicts. If conflicts exists the load operation will fail for all new modules that conflict with the already loaded modules. The entire *Create Federation Execution* or *Join Federation Execution* call will fail if any of the supplied FOM modules introduces a conflict.

8. General Best-practice Principles for FOM Module Development

When you get started with developing FOM modules the following are important things that you need to think of:

Don't skip the Identification table. At some point in time somebody else will probably need to know the origin of your FOM module. This type of data is stored in a well-structured way in the Identification table.

Clearly state the purpose of the FOM module. Why are you developing this module and what is its main purpose? What is the reason for putting this particular set of objects and interaction into a separate module? Is there a particular scope of expected reuse? Is there a particular type of federations that will benefit from it? Or is there a certain group of experts that will be responsible for developing and maintaining it?

Extend reference FOMs using separate modules. If your federation is based on an open or in-house standard FOM module, for example the RPR FOM, do not modify this module. Put all your project-specific extension in a separate FOM module. When your federate is later reused in another federation it is easy to analyze what particular extensions that were used in different projects. If you

Table	Principle	FOM Module A sample data	FOM Module B sample data	Result
Identification	Not combined	Version="2.0"	Version="1.6"	(nothing)
Object classes	Union of class tree	ObjectRoot Vehicle Car	ObjectRoot Vehicle Aircraft	ObjectRoot Vehicle Car Aircraft
Interaction classes	Union of class tree	InteractionRoot RadioMsg	InteractionRoot Explosion	InteractionRoot RadioMsg Explosion
Data types	Union	SpeedInteger AngleFloat64	SpeedInteger CountryEnum	SpeedInteger AngleFloat64 CountryEnum
Dimensions	Union	Country Airline	Country CargoType	Country Airline CargoType
Time Representation	Must be equal if present	HLAinteger64Time	(Not provided)	HLAinteger64Time
Synchronization Points	Union	ReadyToRun	StageB	ReadyToRun StageB
User-supplied tags	Must be equal if present	Update/Reflect= "HLAASCIIString"	Update/Reflect= "HLAASCIIString"	Update/Reflect= "HLAASCIIString"
Transportation Types	Union	LowLatency	Secure	LowLatency Secure
Switches	Must be equal if present	AutoProvide= "Enabled"	AutoProvide= "Enabled"	AutoProvide= "Enabled"
Notes	Union	Note1	Note3	Note1 Note3

Figure 7: An Introduction to Combining FOM Modules

come up with some generally useful extension you may also consider feeding this information back to the reference FOM module development group.

Don't redefine concepts. If you define the same concept, such as a class, a parameter or a data type, in several FOM modules the definitions needs to be identical. If you for example define the data type "Altitude" as a 32 bit little-endian integer, describing the altitude in meters, other modules shall either give exactly the same definition or shall provide no definition at all. The same applies to object classes with attributes, interactions with parameters, switches, time representations, synchronization points, user defined tags and more. A later section of this paper describes what problems you may run into when redefining concepts.

Think twice before repeating definitions. In general it is not a good idea to repeat definitions, such as an object class or a data type, from one FOM module in another module. There are, however, some exceptions. An acceptable reason for repeating concepts is when you have two modules that both need a particular data type and that sometimes will be loaded into the same federation and sometimes used separately. At the same time you introduce a risk that these definitions will deviate over time, making the FOM modules incompatible.

Prefix your Notes with the module name. If you develop a module that will be called "Flight" then consider naming the notes in the notes table "Flight -1", "Flight-2", etc. When several modules are merged for various purposes you will not need to renumber them across the FOM.

Don't forget the Switches table. The RTI requires certain runtime switches when it creates the federation execution. This means that you need to provide a FOM module that contains the Switches table in the Create Federation Execution call. In fact, this is really the only FOM data that is required when creating the federation execution since all the required and predefined concepts in the HLA standard is automatically provided by the RTI in the HLAstandardMIM FOM module. Read more about this module later in the next section of this paper.

Avoid putting the Switches table in a reference FOM. This will prevent federations from using other switches settings without modifying the reference FOM. During federation integration and testing it is useful to modify certain switches for debugging and tuning purposes.

Switch	Suggested default
autoProvide	True
conveyRegionDesignatorSets	False
conveyProducingFederate	False
attributeScopeAdvisory	False
attributeRelevanceAdvisory	False
objectClassRelevanceAdvisory	False
interactionRelevanceAdvisory	False
serviceReporting	False
exceptionReporting	False
delaySubscriptionEvaluation	False
automaticResignAction	CancelThenDeleteThenDivest

Figure 8: Suggested Default Switches

9. The Predefined HLAstandardMIM Module

There are a number of OMT items that always need to be present in a federation. These are contained in the HLAstandardMIM module, where MIM stands for Management and Initialization Module. The most important things are:

The **root classes** for objects and interactions, called HLAobjectRoot and HLAinteractionRoot. You will create your own classes as subclasses of these roots.

The **predefined data types**. Some of them, like HLAUnicodeString, can be used directly for the attributes and parameters that you intend to model. Some others are Basic Data types, like HLAinteger32BE, that cannot be used directly but can instead be used for defining your own data types.

The **Management Object Model** (MOM). These are object and interaction classes that can be used to monitor and control the federates and the federation. These classes are typically not used in more basic federates, but may still be used in simple federations where common federation management tools are used. The most commonly used object class here is the HLAfederate. By subscribing to attributes of this class you will get information about which the other currently joined federates in your federation are. The HLA standard provides an extensive specification of the MOM.

The HLAstandardMIM module is provided automatically for you by the RTI. This means that you don't need to load it when creating the federation execution. The only exception to this is for advanced users that want to extend the MOM.

10. Types of FOM modules

Formally, there are two types of modules:

Standalone modules that builds upon concepts in the HLAstandardMIM only or that don't build upon any other, user-provided module at all.

Dependent modules. These modules are add-on modules that depend on concepts defined in other modules.

The type of module can be indicated in the identification table of the FOM, as can be seen in Appendix A.

There are two additional relationships that are not part of the standard but still useful to understand:

Imagine a module that uses the Start interaction from the RPR FOM. To be useful without the RPR FOM module it repeats this interaction. This FOM module is said to be **compatible** with the RPR FOM module without being dependent. One trivial case when this happens is when modules have nothing in common. The more interesting case is when they contain identical (repeated) definitions of the same concepts (such as data types)

Imagine a FOM module that only contains a Warning interaction. A federate is supposed to send this interaction whenever it detects an aircraft from the opposing side. The use of certain other FOM modules is assumed in the federation agreement. In this case the module is standalone but the federation agreement builds upon several modules. These modules can be considered to be **related**.

11. Switches

The FOM contains a table with runtime switches for the RTI. This table has to be provided in the Create Federation Execution call, or to be more exact: It has to be present in at least one of the FOM modules that are supplied.

11.1 Suggested default values for switches

Here are some suggested default values for these

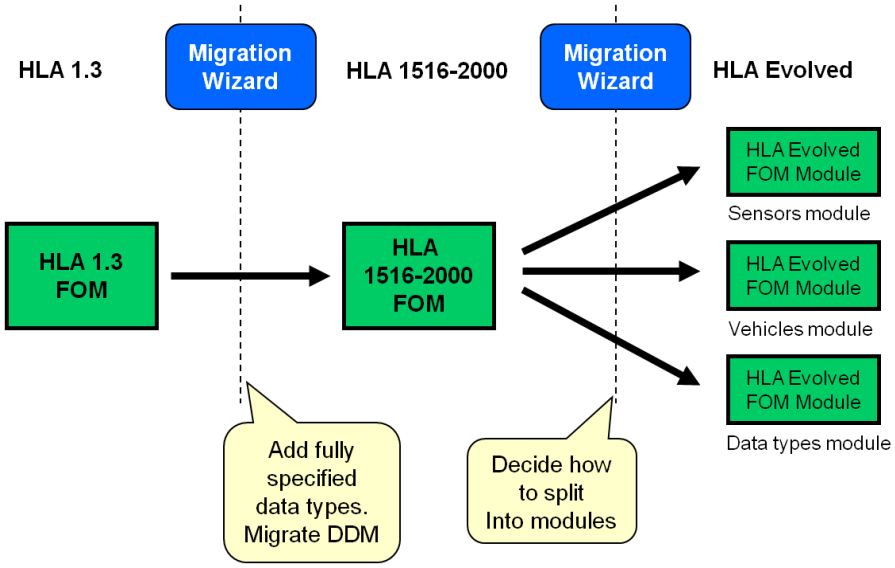


Figure 9: Migrating a FOM to HLA Evolved FOM Modules

switches, see figure 8. Note that these are just suggestions. You need to investigate the design patterns of your federation as well as the service usage of participating federations to find out exactly what's right for you.

The **autoProvide** switch makes the RTI request that attribute updates are provided when a new federate discovers an object instance from another federate. To optimally support late joiners (including federates rejoining after a temporary fault occurred) this switch should be set to true. To fully support late joiners all federates should also be required to respond to the Provide Attribute Values callback.

The **conveyRegionDesignatorSets** switch makes the RTI convey the DDM information for example for attribute updates that are delivered to a federate. This may be useful for debugging. In many cases you may simply want to set this to false.

The **conveyProducingFederate** switch makes the RTI convey the ID of the federate that provided for example an attribute update or an interaction. This is useful for example for logging data from one particular federate. For less advanced federations you may want to set this to false.

There are four **Advisory** switches. They control whether the RTI should send some extra callbacks to a federate indicating whether the data that is produced will actually be used by any other federate. This may be a very powerful base for optimizing more advanced federations. For less advanced federations you may want to set them to false.

There are two **Reporting** switches that may be used to inspect how the RTI is being called by federates and what exceptions that occur. For less advanced federations you may want to set them to false.

The **delaySubscriptionEvaluation** switch enforces a very strict behavior on the RTI. The RTI will need to check whether a certain type of data is still subscribed just before it is delivered from the Local RTI Component (LRC) to the federate code. This may have changed from the moment the data was sent from another, producing federate. For many federations you may want to set this switch to false.

The **automaticResignAction** switch controls the behavior when a federate is lost due to a fault. The RTI will then perform a resign on behalf of the lost federate, using this resign action. To maximum "clean up" after the lost federate the value CancelThenDeleteThenDivest is suggested.

11.2 Strategies for providing the switches

We suggest using one of the following two strategies:

Strategy 1: Provide the switches table as part of the most general FOM module that is provided in the Create Federation Execution call. This is typically the FOM module that provides the base concepts for a domain. This strategy should however be avoided if a reference FOM module is used for basic concepts. Do not put a Switches table in a reference FOM since such a FOM should be read-only and you may want to experiment with the switches for debugging purposes.

Strategy 2: Provide the switches table in a separate FOM module. This is the most general approach but it may be overkill for more basic federations.

12 Errors and Redefinitions

What if you accidentally redefine a concept that already exists in another module? You may encounter one of the following situations:

A) The RTI will refuse to load the FOM module that contains the redefinition of an already loaded concept. An example of this is

when two different modules define different attributes for the same class. This error is encountered when the second of the two conflicting modules are loaded into the Federation Execution. Note that he RTI only looks at a subset of the FOM module data so not all conflicts will be discovered by the RTI.

B) Some federates will crash or behave incorrectly. Imagine if one FOM module defines the data type Altitude as having a 32 bit integer representation and another module defines the same data type as having a 64 bit integer representation. Assuming that different federate developers looked at different modules during the development a crash will most likely occur. The federate that received a 32 bit integer but expected a 64 bit integer will fail to decode it.

C) More or less subtle errors and inconsistencies will occur in the simulation. Imagine if one FOM module defines the data type Altitude and states the units to be meters and another FOM module states the units to be feet. Imagine that different federate developers develop their systems according to these two different definitions. This mistake may be difficult to discover until the simulation output is carefully examined.

D) For a few trivial cases, like misspellings in the semantics field of a data type, nothing bad will happen.

13. Migrating FOMs to HLA Evolved

There are COTS tools readily available that facilitates the FOM migration. This process is shown in Figure 9.

If you currently have an HLA 1.3 FOM you are recommended to first migrate it to the HLA 1516-2000 format. The biggest difference here is if you are using DDM. This has been redesigned and standalone Dimensions have replaced Routing Spaces. During this conversion process you are also highly recommended to add fully specified Data Type descriptions.

The step from HLA 1516-2000 to HLA Evolved is quite simple using a COTS tool. All you will need to do is to specify exactly which tables as well as which object classes and interaction classes that you want to go into which module. The rest of the conversion is automated. While this is technically simple and large FOMs can be migrated in minutes, the reader is still recommended to carefully consider the design recommendations provided in this paper.

14. More Areas to Explore

This paper is intended to give an introduction to FOM modules, not a complete specification. There are a number of areas for the reader to further explore:

A) Study the HLA Evolved specification to understand the exact specification of FOM modules in general and FOM module merging in particular.

B) Study the Base Object Models (BOM) that provides a structure workflow from conceptual models to FOMs and FOM modules.

C) Since FOM modules can be added when a federate joins, it is possible to extend the FOM of already running federation over time. Figure out some new opportunities that this enables.

D) In a large FOM project you may want to put your data types in a separate FOM module. What are the pros and cons of this approach?

E) When you load FOM modules into a federation using the Create or Join call, the RTI will check that the FDD part of the FOM modules can be merged. Study the HLA specification to find out exactly which parts that the RTI checks.

F) Study the HLA Evolved Interface Specification and the Create Federation Execution call in particular to find out how you provide a user-extended HLAstandardMIM.

15. Conclusions

This paper has provided an introduction to the practical usage of FOM modules, including several examples.

FOM Modules is one of the most important additions to the HLA standard in the HLA Evolved version. They enable us to develop and reuse FOMs in a more modular fashion, to separate temporary adaptations from reference FOMs and to split the development work between different teams. In the long run they will also enable us to reuse partial federation agreements and federation design patterns.

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 1516-2009, High Level Architecture (HLA)", www.ieee.org, To be published.

[4] 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.

[5] Björn Möller, Björn Löfstrand, Mikael Karlsson. "Developing Fault Tolerant Federations using HLA Evolved" Proceedings of 2005 Spring Simulation Interoperability Workshop, 05S-SIW-048, Simulation Interoperability Standards Organization, April 2005.

[6] 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.

[7] 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.

[8] 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. Joint paper with SAIC, AEgis Technologies, Carnegie Mellon University/Software Engineering Institute and Johns Hopkins University/Advanced Physics Lab.

[9] IEEE: "IEEE 1516.3-2003 IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)", www.ieee.org

[10] SISO, "BOM Template Specification", SISO-STD-003-2006, SISO, 31 March 2006.

[11] Björn Möller, Paul Gustavson, Bob Lutz, Björn Löfstrand. "Making Your BOMs and FOM Modules Play Together", Proceedings of 2007 Fall Simulation Interoperability Workshop, 07F-SIW-069, Simulation Interoperability Standards Organization, September 2007

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

[13] Fred van Lieshout, Ferdinand Cornelissen, Jan Neuteboom, Björn Möller. "Simulating Rail Traffic Safety Systems using HLA 1516", Proceedings of 2008 Euro Simulation Interoperability Workshop, 08E-SIW-069, Simulation Interoperability Standards Organization, June 2008.

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.

BJÖRN LÖFSTRAND is Manager of Modeling and Simulation Services at Pitch Technologies. He has been involved in the development of several M&S standards and has been working with HLA federation development and tool support since 1996. Björn holds an M.Sc. in Computer Science from Linköping Institute of Technology. Recent work includes developing FOM and Federation Design for NATO Education and Training Network.

Appendix A: The Tiny FOM Module

```
<?xml version="1.0" encoding="UTF-8"?>
<objectModel xmlns="http://www.sisostds.org/schemas/IEEE1516-2009"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.sisostds.org/schemas/IEEE1516-2009
         http://www.sisostds.org/schemas/IEEE1516-DIF-2009.xsd">
  <modelIdentification>
    <name>Small railroad FOM</name>
    <type>FOM</type>
    <version>1.0</version>
    <modificationDate>2009-08-12</modificationDate>
    <securityClassification>Unclassified</securityClassification>
    <purpose>Train training</purpose>
    <applicationDomain>Railroad</applicationDomain>
    <description>Basic train data exchange for analysis applications</description>
    <poc>
      <pocType>Primary author</pocType>
      <pocName>Björn Möller</pocName>
      <pocOrg>Pitch Technologies</pocOrg>
      <pocTelephone>0046 13 13 45 45</pocTelephone>
      <pocEmail>info@pitch.se</pocEmail>
    </poc>
    <reference>
       <type>Standalone</type>
       <identification>NA</identification>
     </reference>
  </modelIdentification>
  <objects>
    <objectClass>
      <name>HLAobjectRoot</name>
      <objectClass>
        <name>Train</name>
        <sharing>PublishSubscribe</sharing>
        <semantics>Generic train</semantics>
        <attribute>
          <name>Speed</name>
          <dataType>SpeedInt64</dataType>
          <updateType>Conditional</updateType>
          <updateCondition>On change</updateCondition>
          <ownership>NoTransfer</ownership>
          <sharing>PublishSubscribe</sharing>
          <transportation>HLAreliable</transportation>
          <order>TimeStamp</order>
          <semantics>Speed of the train</semantics>
        </attribute>
      </objectClass>
    </objectClass>
  </objects>
  <interactions>
    <interactionClass>
      <name>HLAinteractionRoot</name>
    </interactionClass>
  </interactions>
  <dimensions/>
  <tags/>
  <transportations/>
  <dataTypes>
```

```
    <basicDataRepresentations/>
    <simpleDataTypes>
      <simpleData>
        <name>SpeedInt64</name>
        <representation>HLAinteger64BE</representation>
        <units>mph</units>
        <resolution>1</resolution>
        <accuracy>5</accuracy>
        <semantics>Simulated speed</semantics>
      </simpleData>
    </simpleDataTypes>
    <enumeratedDataTypes/>
    <arrayDataTypes/>
    <fixedRecordDataTypes/>
    <variantRecordDataTypes/>
  </dataTypes>
</objectModel>
```

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 in-

troduce 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 RtiFactoryFactory. 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.

Processes and Tools for Management and Reuse of FOM Modules

Björn Möller
Fredrik Antelius
Martin Johansson
Björn Löfstrand
Åsa Wihlborg

Pitch Technologies
Repslagaregatan 25
S-58222 Linköping
Sweden

bjorn.moller@pitch.se
fredrik.antelius@pitch.se
martin.johansson@pitch.se
bjorn.lofstrand@pitch.se
asa.wihlborg@pitch.se

Keywords:

HLA Evolved, OMT, FOM Modules, FEDEP, Reuse, Tools

ABSTRACT: *One of the most important new features of HLA Evolved is FOM Modules. The FOM is used to describe the information that is exchanged in a federation. By making the FOM modular it is possible to focus on certain aspects of the data exchange. This makes it possible to identify, develop and isolate more general or more specific aspects of the data exchange. Examples of more general and thus more reusable modules are commonly used data types or federation management interactions. Less reusable FOM modules are for example project-specific platform extensions to the RPR-FOM.*

The first part of the paper covers processes for the development and maintenance of reusable FOM modules. These can be developed and reused in a "top-down" manner within domains (i.e. defense, space, industry, etc), groups of organizations (e.g. SISO, NATO), organizations (i.e. companies, defense components) and individual projects. Important success factors here are a well-defined lifecycle process, proper management support and a use case and test driven development approach.

It is also possible to develop reusable FOM modules from a more technical "bottom-up" perspective. Useful components like FOM elements are sometimes reused within and across federations, often even beyond the planned life span. The main reusability criteria here is "the survival of the fittest".

The second part of the paper describes how a specialized tool can be used to develop and maintain a project consisting of several FOM modules for use in a particular federation.

These modules need to be inspected, understood and verified both against the HLA Evolved standard and against each other. Since FOM modules can build upon each other it is important that a tool can help the user maintain compatibility and avoid undesirable dependencies between modules.

When managing FOM modules it is important to understand what role each FOM module plays from a reuse perspective. Is it a highly standardized module or a temporary project development? This affects which modules that should be adjusted when consistency and compatibility issues are discovered. It affects several aspects of refactoring across modules. Last but not least it affects how a tool can provide "best practices" assistance to a new user. A comparison is also made between maintaining FOM modules using general-purpose XML tools versus a specialized FOM module tool.

Finally some thoughts on the above processes and tools are given, based on the on-going work with the NATO Snow Leopard federation and other practical applications.

1. Introduction

Whenever you connect two or more systems to exchange data there will be an information exchange data model. This model may be explicit or implicit but it will always be there. For typical business systems this model may be focused on providing a simple service with a well-known set of outcomes, for example verifying a credit card transaction. For simulation systems that are required to interoperate, the information exchange may include a large number of entities and interactions, possibly describing an entire battlefield.

A large number of simulators may need to consume information from each other. The resulting effect of a small state change may, from case to case, be minimal or massive and it may sometimes be hard to predict for the original information producer. For the simple interaction between business systems a two-part point-to-point data exchange may be sufficient. For simulation systems with many participating systems on the other hand a common data bus is the optimal solution.

1.1 Challenges with hard-coded domain models

When exchanging data the most obvious approach is to create a network protocol. This specifies where in each exchanged data block each domain property should be stored, for example DIS [1]. Certain bytes in the exchanged packets may describe the marking or the position of an aircraft, in effect hard-coding the protocol to a particular solution approach for a certain domain. This makes it convenient to adapt each new simulator to the protocol since both the format and the content of the protocol are well-known in advance. The problem here is that there will always be variations in the requirements and that requirements will grow over time. For slight variations a few non-standard packets can be introduced. For applications with different requirements the protocol may not be useful at all. It is also difficult to introduce more advanced simulation services since each simulator may need to correctly implement them.

1.2 Separating out domain information

General purpose protocols, like TCP/IP [2], FTP [3], HTTP [4] and SMTP [5] have generally been very successful. Today they form the basis of the Internet and any corporate network. These typically standardize the technical level of communication. The actual domain data, like the layout and text of a web site are described on a higher level of communication. By separating the lower level protocol from the domain data each user in various application domains is given the power to describe his domain information.

The trend today is to capture the information exchange data model on a higher and more flexible level than the network packet level. This can be seen in the FOMs of the HLA standard [6], the LROMs of the TENA framework [7], the WSDL approach of Web Services [8], the Topic of the DDS [9] architecture and more. A parallel can also be drawn on the Variable Message Format (VMF) protocol of the Link-16 [10] family.

2. The HLA FOM

The information exchange model of HLA is called the Federation Object Model (FOM). It is based on the Object Model Template, which is one part of the HLA standard. The FOM describes both the information that is exchanged at runtime as well as the usage and parameters of a number of additional HLA services. It is important to understand that the FOM is only one part of the "Federation Agreement". The Federation Agreement is the document where you find descriptions of overall federation purpose, expected sequences of interactions, which federates that are producers and consumers of certain data, networking, etc.

2.1 Content of a FOM

The FOM can contain data in a number of different tables. A typical starter FOM usually includes the following tables.

Identification table, describing things like the purpose, author and version of the FOM.

Object Classes and Attributes tables, describing the persistent entities (like aircrafts) that are shared between the federates

Interaction Classes and Parameters tables, describing short-lived data like commands or radio traffic (i.e. events) that needs to be exchanged.

Data Types table describing the technical format and interpretation of the attribute and parameter data.

More advanced FOMs may also include the following:

Additional Federation Execution Data, like the **Dimensions table** (for DDM data routing), **Synchronization Points table** (for synchronizing the federation) and **User Supplied Tags table** (specifying the data format of extra parameters in certain HLA Services).

Additional Infrastructure Settings like the **Time Representation table**, the **Update Rate table**, the **Transportation Types** table and the **Switches table**.

For additional documentation across the tables it is also possible to attach a number of notes using the

Format	Technical format	Specified using	New features
HLA 1.3	BNF (”LISP” style)	BNF + textual specificatition	-
1516-2000	XML	DTD + textual specification	Routing Spaces table replaced with Dimensions. New table: Data types.
1516-2010	XML	Three XML Schemas + textual specification	Modular and extendable format. New tables: Update Rates, Services Utilization. Extended tables: Transportation Types, Identification

Figure 1: The evolution of the HLA OMT format

Notes table. There is also a **Services Utilization Table**.

2.2 Evolution of the FOM format

The FOM follows the OMT format of HLA. This format has evolved over time as shown in Figure 1. Two major driving factors can be noticed. First of all, new tables have been added or existing tables have been extended to meet new technical requirements. Secondly, the technical format has gradually been adapted to follow the most recent XML format descriptions.

2.3. Maintaining a FOM for multiple HLA standards

It is possible to convert FOMs using the older format to a newer format with automated tools. In some cases design decisions needs to be made, like mapping Routing Spaces to Dimensions or splitting a FOM into modules. This can technically be solved with a wizard, allowing for user input during the conversion process.

When maintaining a FOM for multiple standards it may be more convenient to keep the original FOM in a newer format since this is in many respects a superset of the older format. Data can then be converted back and new types of information can simply be excluded.

3. FOM Modules

The suggestion to make the FOM modular was the last comment in the last SISO comment round for the new HLA 1516-2010 standard. Still this idea had been around and discussed since the very first OMT specification in the 90’s.

The Modular FOM approach is very simple. Each FOM module describes a certain aspect of the information exchange. A FOM module can contain whatever FOM data that is required for its purpose, for example just a few object classes with attributes and corresponding data types. Several FOM modules can be combined into the final FOM to meet the requirement of a federation. You may for example combine a module describing vehicles with other modules describing radio communication, federation management and data logger control. The FOM data from these modules are then merged producing the union of the modules.

FOM modules make the development and reuse of FOM data more powerful. In many cases it makes FOM development easier since different development groups or parts of the development cycle can focus on particular areas of the FOM development.

For an extensive introduction to FOM Modules the paper “Getting Started with FOM Modules” [11] is strongly recommended.

3.1 Use cases for FOM modules

As described in [11], FOM modules can be used for several purposes:

You can have **different working groups** develop different parts of a FOM in a more convenient way. You may for example have a radio specialist group develop the “Radio FOM module” while the aircraft specialists develop the “Aircraft FOM module”.

You can put **extensions to a reference FOM** in a separate FOM module. This will prevent you from getting modified, “non-standard” versions of the reference FOMs. More importantly, when you build new federations using federates that use extended reference FOMs, it is easy to inspect what extensions that have been made and possibly to merge them.

You can achieve **extended reuse** of some aspects of a FOM. If you want to promote a standardized way to start and stop all of your federations, irrespective of domain, you may put these interactions in a separate FOM module.

You may also **add more FOM modules to an already executing federation**, thus extending the scope of the FOM during runtime.

Restaurant FOM	Sample FOM that is included in the HLA standard, available in HLA 1.3. 1516-2000 and 1516-2010 format. This FOM has been modularized as a sample in the product described below. It will be made publicly available on the SISO HLA Evolved PSG reflector in Sep 2010
RPR FOM	FOM mainly targeted at real-time platform simulations, standardized by SISO, matches the DIS data model. It is a good candidate for splitting into FOM modules. Some early work on modularization has been done as part of the BOM [12] standard, as described in another paper [13]
LINK-16 BOM	The SISO LINK-16 BOM is essentially a FOM module for LINK-16 [14] data exchange.
NASA	NASA has performed some early FOM module prototyping for the Constellation program as described in another paper [15]
P2SN FOM	Persistent Partner Simulation Network, previously Partnership for Peace Simulation Network. Originally in HLA 1516-2000 format, now in HLA 1516-2010 format. Uses the RPR FOM as one module and has both more general and more specialized FOM modules as described in another paper [16]
NETN FOM	NATO Education and Training Network FOM is expected to supersede the P2SN FOM above and will thus be modular.

Figure 2: Some efforts related to FOM modules

3.2 Some early modular FOM efforts

The HLA Evolved standard has recently been completed but there are already some early use cases for the modular FOM and some related efforts. Some examples are shown in Figure 2.

4 Processes for FOM module development

From the point of view of a particular federation the FEDEP/DSEEP [17] process describes at what stage the FOM needs to be developed. In this paper we look at the overarching perspective of reusing FOM modules across projects. This is further divided into how reusable FOM modules are produced and how they are later reused in other projects.

4.1 Developing standardized and reusable FOM modules top-down

Assume that a certain community wants to increase the potential for reuse of interoperable simulators. One of the most important efforts towards that goal is then to agree on a common, standardized FOM module. Since requirements will vary from federation to federation a good starting point is to include the most common shared objects and interactions in their domain. More specialized aspects can be covered in project-specific FOM modules. As many simulators in the community are adapted to the common FOM module they will have a basic level of interoperability.

Such FOM modules can be developed on several levels as shown in Figure 3.

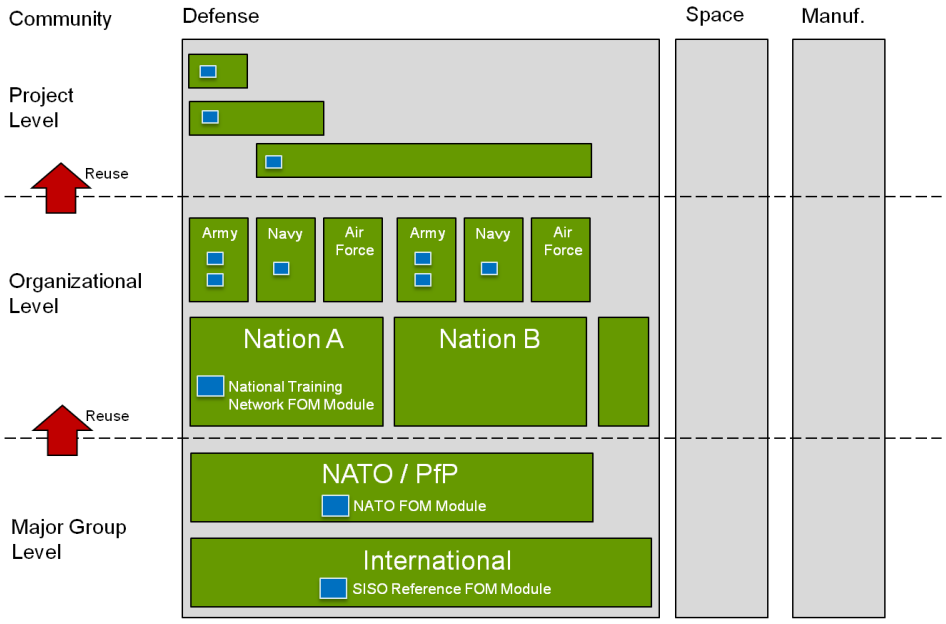


Figure 3: Developing reusable FOMs top-down

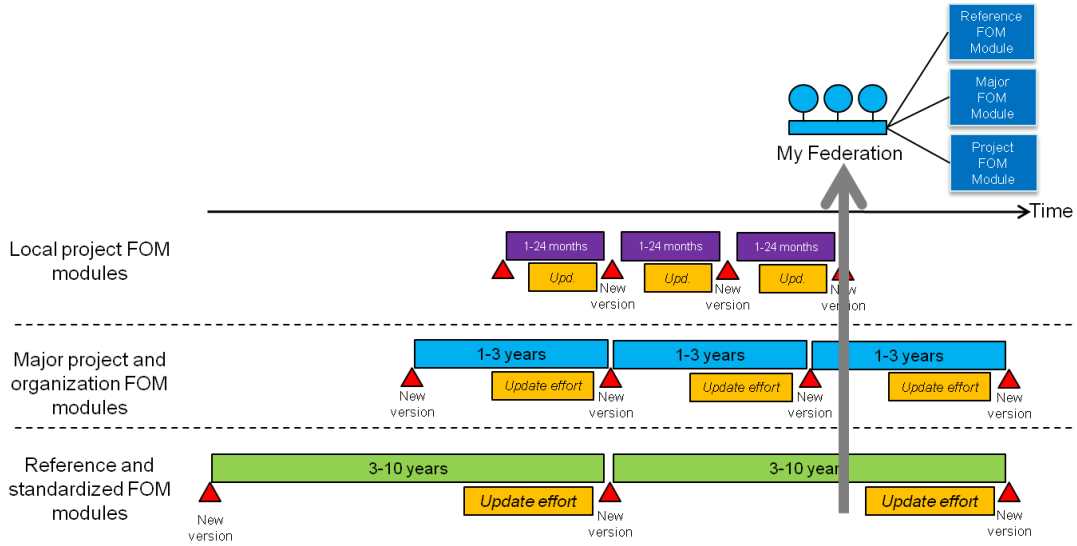


Figure 4: Reusing FOM modules with different lice cycles

On the overall community or domain level (defense, space, energy, manufacturing, etc) FOM modules can be developed in open international standardization forums, like SISO. The RPR FOM is an example of a FOM in the earlier HLA formats that is developed this way. Major groups with common needs, for example NATO, may also choose to develop common modules.

On more specific levels reusable FOMs can be developed within nations, within defense components or within companies.

Some important success factors when developing FOM modules are:

A clearly defined goal and a common requirements picture for the FOM module to be developed. Otherwise a consensus on a solution may never be reached.

A well-defined development process where new versions are developed, documented and released in a clear process that is transparent to the participants. Otherwise numerous of intermediate and ill-understood versions of the FOM will be used, actively preventing rather than enabling interoperability.

Management support for the development activities, for participants and for the resulting FOM module. If this cannot be achieved the development resources may be under-critical and the FOM will never be finalized or put into production.

Use-case and test-driven development. Practical experiences show that a suggested FOM solution that looks good on paper may contain minor glitches that makes it hard or impossible to use. Practical tests are the only solution to this. This process is unfortunately seldom documented in papers. An example from the DIS world of how a data exchange model has been adjusted after testing is the Directed Energy PDUs [18].

4.2 Developing reusable FOM modules bottom-up

It is also possible to develop reusable FOM modules from a more technical "bottom-up" perspective. Useful components like FOM elements are sometimes reused within and across federations, often even beyond the planned life span. The main reusability criteria here is usually "the survival of the fittest".

4.3 Reusing FOM modules

When reusing FOM modules in a federation a developer will typically start with some reference FOM modules. These can then be extended with locally produced FOM modules and extended with some project-specific FOM modules. In this case dependencies will typically be introduced. It is now important to look at the life-cycle of different modules. As shown in Figure 4, reference FOM modules will probably have a version life-cycle in the range of years whereas the project may update a FOM module every month. Proper configuration management of FOM modules is strongly recommended here.

4.4 Resulting requirements for a FOM module tool

Some important requirements for a FOM module editing tool that can be derived from the above are:

1. A tool should make it easy to **combine** different FOM modules that cover different information exchange requirements of the federation. It should be easy to get an **overview** of the combined modules and to **discover, analyze and fix any mismatches.**

2. A FOM module tool should enable a developer to **build upon standardized FOM modules**,

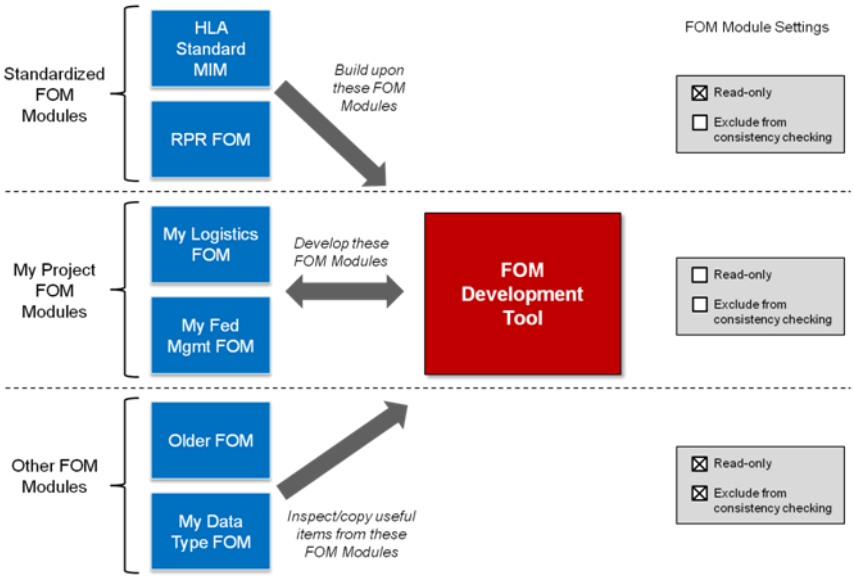


Figure 5: Composing a FOM from existing modules

add his own local extensions and also to reuse components from other, less related projects, as shown in Figure 5.

3. Each FOM module designer need to clearly understand which of the FOM modules in a project that he is responsible for updating and which that are maintained by other designers. A tool should support the developer in **updating only his own modules** and keep other read-only.

4. It is important to understand which FOM modules that are allowed to depend on others modules and which these modules are. A FOM module tool should assist in **maintaining proper dependencies**.

5. In many cases a user wants to work with existing modules just to copy data types or object classes that are generally useful. This module may never be intended to be used in the final FOM. A FOM module tool should make it easy to **reuse smaller components of older FOMs** that in their entirety may be less reusable.

5. A Next Generation FOM editing tool

A commercial tool for developing FOM modules, Pitch Visual OMT version 2.0, has been developed based upon the above analysis. It builds upon older versions that supported HLA 1.3 and HLA 1516-2000 FOMs. This versions is a complete reimplementation with HLA 1516-2010 FOM modules as the native file format. Still it supports many file formats of older HLA versions. Pitch Visual OMT 2.0 is planned to ship September 2010.

The overall design resembles a traditional graphical programming environment as can be seen in Figure 6. A project is composed from several FOM modules, including a standard or custom HLAstandardMIM. These modules are listed to the left and en-

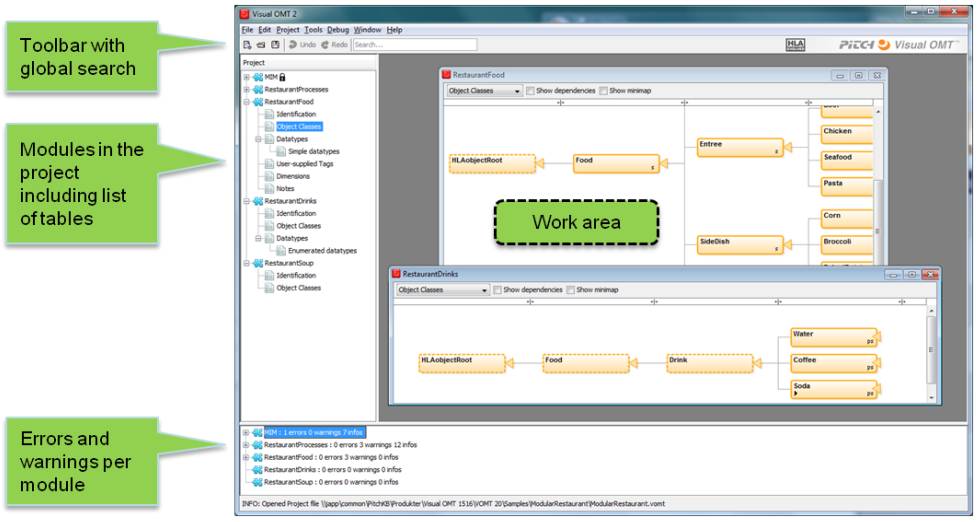


Figure 6: Overview of the Visual OMT 2.0 user interface

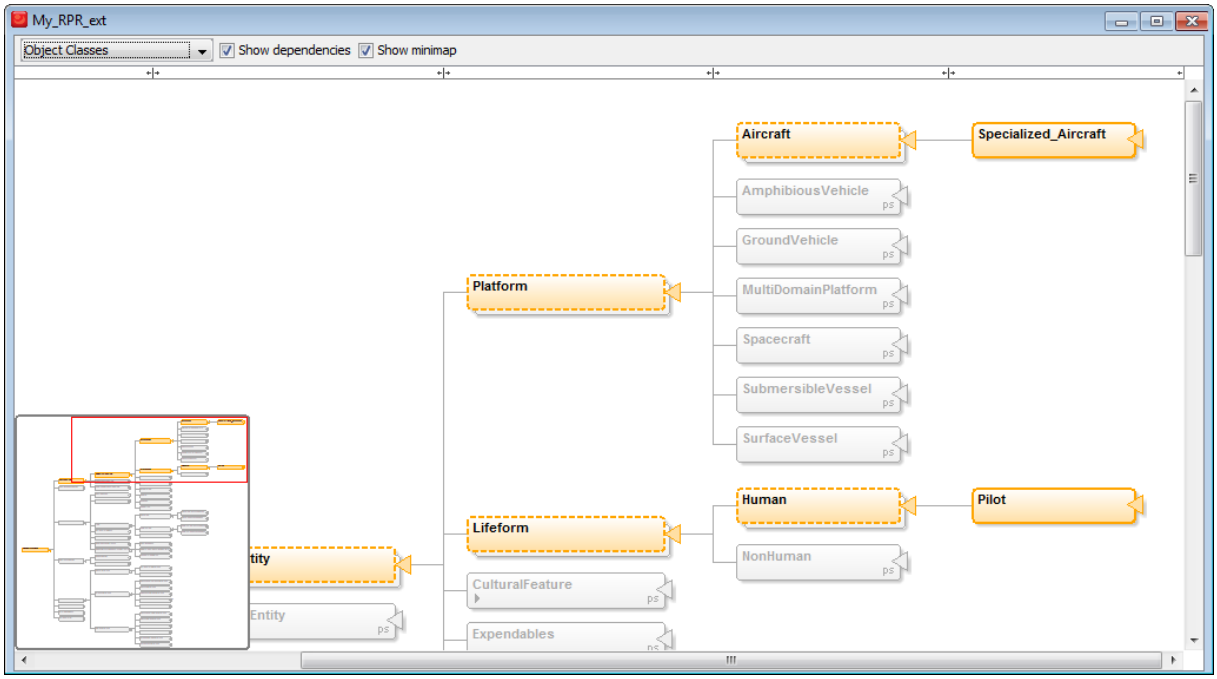


Figure 7: Object class view in Visual OMT 2.0

ables the user to inspect which tables that are included in each module. A padlock icon indicates that a module is read-only.

FOM modules are opened in the central work area. By double-clicking on a particular table in a module a developer can jump directly to that table, inspect FOM data, edit and use drag-and-drop.

The entire project is continuously analyzed in the background. Issues (errors, warnings and tips) are presented in the bottom area as a "To Do" list. There is a traditional toolbar at the top. One of the most interesting features here is the Global Search, as described later in this paper.

5.1 Visualizing combined FOM modules

Most FOM modules contain classes and subclasses for Object Classes and Interactions. The combined modules thus may result in a very large class hierarchy. The classes of each FOM or a set of dependent FOMs can be visualized as a tree graph in Visual OMT.

Modularized FOMs build upon each other and often extends classes in reference FOMs. The newly created subclass in such an extending FOM will have all its super classes defined as scaffolding classes. A scaffolding class is a way for a FOM designer to place a new class inside an existing class hierarchy without the need to duplicate all the super classes inside his FOM module. This allows the designer to see the whole chain of objects from HLAobjectRoot to the new class.

However, when looking at only one module it is not possible to know if a class in the current module is also defined in another module, or if the class inherits attributes from a super class in another module. Classes exist in a context dependent on which module it is supposed to merge with. This concept is hard to grasp while only looking at one module at a time. By combining several class trees and displaying them together a more comprehensive view of the context of the classes is created. In a combined view it is easier to see where to place new classes and to see how it fits with already existing classes.

Figure 7 depicts such an combined class tree structure. The colored classes belong to the module currently being edited and the grey classes provide context information. When there are multiple definitions of a class in several modules it is shown as a stack. The dashed lined denotes a scaffolding definition. Here it is easy to see if a scaffolding class has a proper full definition in another module and if any of the definitions clash with other modules. Editing is simplified by allowing drag and drop to rearrange classes. Making it easy to maintain consistency with other modules is also enabled by supplying functionality to copy a full definition to a module and create new classes that automatically get the appropriate scaffolding classes to connect it with the combined class hierarchy.

5.2 Locating information

Working in large FOMs, or rather with a collection of FOM modules where the resulting FOM is large, there are several features that can simplify navigation. In Visual OMT 2.0 there is a global search function which make it possible to search the project in order to find a certain type or concept. An

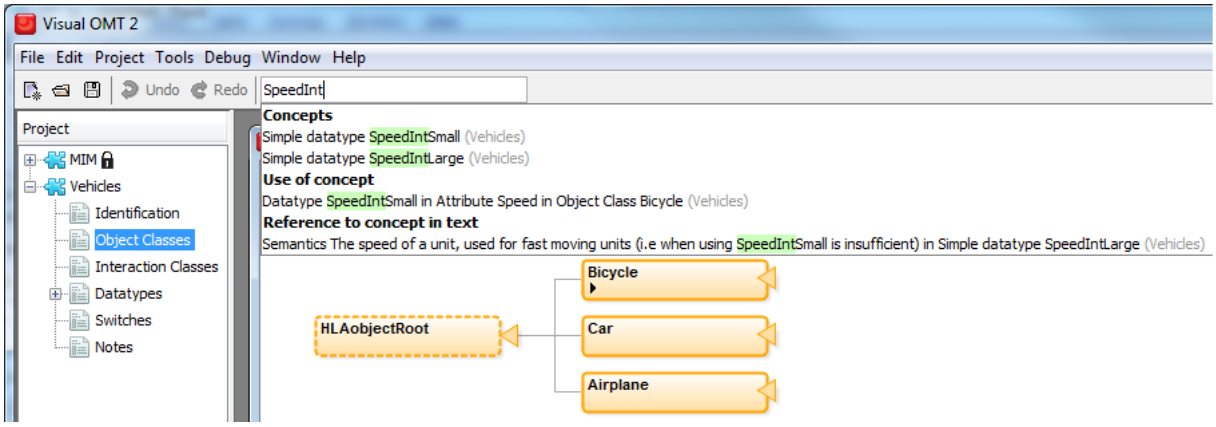


Figure 8: Example of a global search in Visual OMT 2.0

example of a global search can be seen in Figure 8. It is possible to double click on an item in the list to go to its corresponding table. There is also a specialized datatype search used to find the datatype appropriate for attributes and other concepts that uses datatypes. In this search it is possible to sort the resulting datatypes on name, type and other characteristics

To ease navigation in large class trees there is a mini-map showing which part of the graph is currently displayed, visible in the lower left corner of Figure 7. The mini-map also allows for quick navigation. In the main view there is also the possibility to expand and collapse sub-trees to look at only parts of the tree.

5.3 FOM dependencies

When working with FOM modules it is often a good idea to gather information that many FOM modules need in a single FOM module. This single FOM module would contain datatypes, dimensions and other objects shared among the FOM modules. This introduces a dependency between the different FOM modules in which one FOM module is dependent on another and requires that it also is used in a federation.

Another situation which would create a dependency is when a FOM designer uses a reference FOM, such as the RPR FOM, to create a FOM module that extends the reference FOM.

Visual OMT 2.0 helps a FOM designer by allowing him to explicitly define allowed dependencies and then enforces them. Any usage of objects, such as a data type, that's not defined in the module itself, or its dependencies, will generate a warning prompting the user to either define the dependency explicitly or to use another data type.

The concept of dependencies also reduces the amount of data to process when editing large FOMs, since data from modules that are not specified in dependencies will not show up as alternatives when choosing data to use in the new module.

5.4 Read only FOM modules

In larger projects the responsibility for creating the FOM is often split up amongst a group of people so that each person creates a FOM module that together forms the final FOM. This means that from the perspective of one person the FOM modules he is not responsible for should be considered read only, as he lacks the authority to change them. Another case in which a FOM module should be considered read only is when developing an add-on module to a reference FOM such as the RPR-FOM, in which case the RPR-FOM should be considered read only. In cases such as these Visual OMT 2.0 allows the FOM designer to mark a module as read only. A read only module may not be modified but FOM modules that depend on it are allowed to make use of the data in it as described earlier.

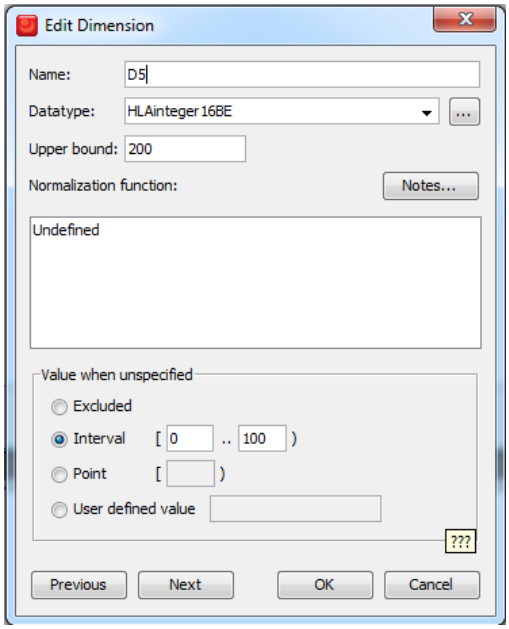
5.5 Sample FOM Module projects

Sample projects makes it easy to get started and demonstrate how different parts of a FOM are defined and how they work together. It also gives examples of how functionality is distributed among the modules. Some examples included in Visual OMT 2.0 are:

Hello Modular World: a very basic sample

Modular Restaurant: based on the sample Restaurant FOM in the HLA Standard. This example illustrates both how one module (Soup) can build upon another module (RestaurantFood) as well as how two independent modules, in this case RestaurantFood and RestaurantProcesses can stand side by side.

Sample RPR FOM Extension. This example may be more difficult to grasp for someone who is unfamiliar with the RPR FOM. Still it represents an example of how many real-world developers would use FOM modules.




```
<dimension>
    <name>D5</name>
    <dataType>HLAinteger16BE</dataType>
    <upperBound>200</upperBound>
    <normalization>Undefined</normalization>
    <value>[0..100)</value>
</dimension>
```

Figure 9: Editing with a dedicated tool versus editing text format

5.6 Help and explanations

A great advantage of a specialized editing tool as opposed to general XML editors is that more support can be added to help the user enter correct input. Even though XML editors can contain some support, there are several things they cannot check and they can never give more help than there is in the XML schema. A specialized tool like Visual OMT 2.0 has "tooltips", help and explanations as well as GUI components that simplifies editing. Instead of having to know by heart which data types there are to choose from, a drop down box show the alternatives. The GUI also help with advanced formatting like Dimension default values, see Figure 9.

5.7 Locating errors in a FOM module

An easy mistake to do when creating a FOM module is to create references to non existing objects. For example, by misspelling a data types name or defining that an attribute should use a data type that has not yet been defined and later on forgetting to do so. In Visual OMT 2.0 the FOM designer is presented with a list of already defined data types when selecting a data type for an attribute or selecting the dimension for an interaction class. This also incorporates the dependency system described in an earlier section. When selecting a data type for an array, not only data types in the current module will be available, but all data types in the current module and its dependencies

As mentioned earlier an easy mistake to do when creating FOM modules is to create a reference to a non existing object. This is of course not the only kind of error that can occur in a FOM module. Visual OMT 2.0 can identify almost 200 different types of issues. These have been divided into three different groups, errors, warnings and best practice. An error is classified as something that would make the RTI unable to successfully load the FOM module such as two object classes with the same fully qualified name but with different definitions. A warning is something that is not correct but the RTI will still be able to load the FOM module, for example a reference to a non existing data type in an attribute. Finally best practice is tips for creating a good FOM, this includes filling out the Identification table and defining semantics for all your objects.

Visual OMT 2.0 continuously analyzes the FOM modules in the project. When issues are found, they are presented in a to-do list at the bottom of the screen as shown in Figure 10. This gives the FOM designer an easy way to identify how many errors exist in the FOM modules as well as a way to quickly go to an error by double clicking on it in the to-do list.

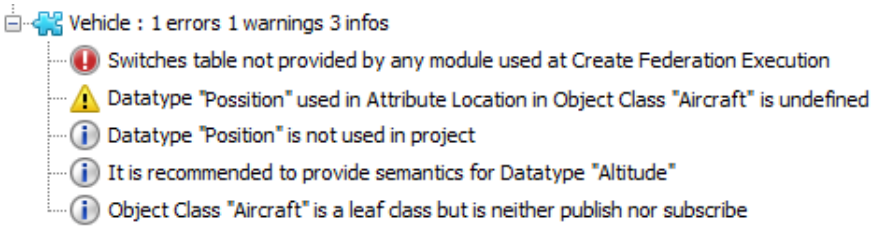


Figure 10: Example of the issues identified in a project in Visual OMT 2.0

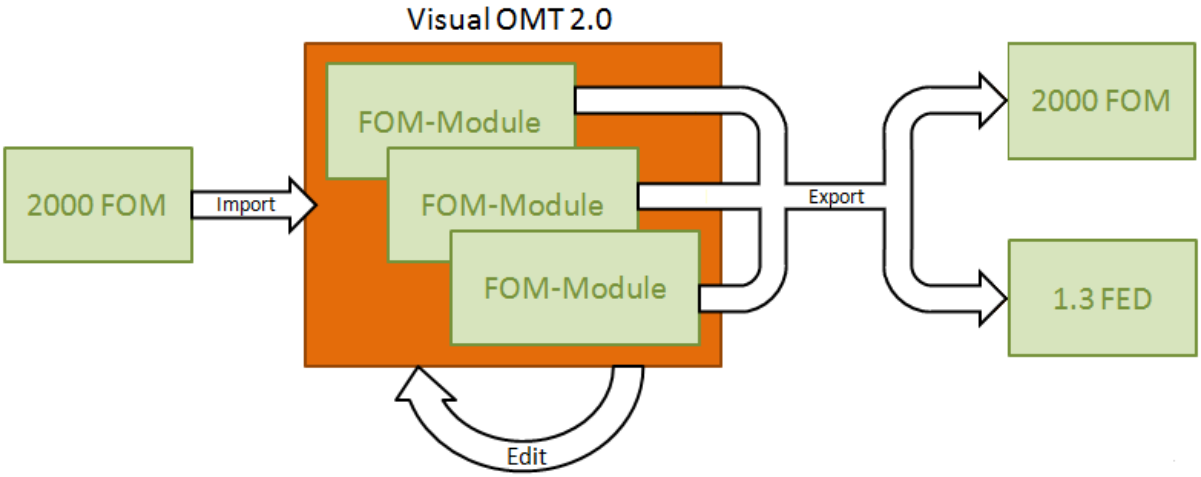


Figure 11: Handling different HLA formats

5.8 Help with module design

Another way in which a FOM editing tool can help a FOM designer is by suggesting good default values. Not only default values for specific things such as the order type of an attribute but design choices like which tables is normally best practice to define in a FOM module. Default values can be controlled per project so that a training oriented FOM uses best-effort transportation by default whereas an analysis FOM defaults to reliable transportation and time stamp order delivery.

Visual OMT 2.0 will also always create an Identification table in a FOM module, although it won't force the FOM designer to actually enter anything useful.

5.9 Reusing FOM data

Reuse of FOM data in a non specific FOM editing tool can be tricky, often involving copy of raw text where care has to be taken to not miss an XML tag and correctly paste the information in the new FOM.

Reusing elements from older FOMs is simple in Visual OMT 2.0. It is possible to open the module and drag and drop entities like object classes or data types to other modules.

If you want to have an older FOM in your project to consult as a reference you can choose to exclude it from the consistency checking in the project since such a FOM probably would introduce multiple errors if combined with the other FOMs.

5.10 Ensuring valid HLA 1516-2010 File format

A fundamental requirement on a FOM editing tool is that it should produce a valid FOM file. A FOM file has a valid OMT syntax if it validates against the DIF XML schema defined by the HLA standard.

In Visual OMT 2.0 this is ensured by using a software component for reading and writing FOM files which is based on this Schema. This also means that Visual OMT 2.0 easily can handle any changes to the FOM format by updating the software component.

5.11 Handling older HLA formats

With HLA Evolved and two older HLA versions now available it is often the case that a FOM needs to be maintained for use in federations using different HLA versions. In Visual OMT 2.0 this can be solved by maintaining one set of FOM modules in the latest HLA Evolved format and then exporting them to older formats. Export to both HLA 1516-2000 and to HLA1.3 FED allows you to run any version of the RTI. An overview of this approach can be seen in Figure 11.

Visual OMT 2.0 also has the ability to import old FOMs from HLA 1516-2000 meaning you can rearrange old FOMs into modules making them easier to understand and maintain.

5.13 Future development

One feature we are looking at adding to Visual OMT 2.0 in the future is refactoring. This would, for example, allow a FOM designer to change the name of a data type and the new name would propagate to all instances where the old name was used.

We are also planning to add more analysis for detecting issues within a project. This could also lead to Visual OMT 2.0 being able to supply the user with a suggested method of resolving an issue and then carry it out if the user accepts it.

More convenient methods for importing and exporting data will also be added. One example is the ability to paste a column of data from Excel into Visual OMT 2.0. This is very useful when defining enumerator values for a new enumerated data type.

Feature	Text Editor	XML Editor	Visual OMT 2.0
Ensures correct syntax	No	Yes	Yes
Help with semantics	No	No	Yes
Ensures correct references	No	No	Within and between modules
Checks best practice	No	No	Yes
Graphical visualization	No	General XML format graph	Object and Interaction class trees
Tabular visualization	No	No	Similar to the HLA standard
Dependency analysis	No	No	Yes, searchable
Search	In one module	In one module	Across several modules
Import/export between formats	No	No	Yes

Figure 12: Comparison between FOM editing options

5.12 A comparison with other editors

Figure 12 summarizes some functional differences between editing options. In addition to this cost may also be considered. A basic text editor like Notepad will of course be a cheap alternative for small projects. Larger projects will want to opt for a dedicated HLA OMT editing tool.

6. NATO Snow Leopard example

This section describes a project where FOM modules are used. It gives a project overview and then shows an example of how a real FOM module looks in Visual OMT 2.0.

6.1 Project overview

The NATO Snow Leopard a.k.a. NATO Education and Training Network (NETN) is based on recommendations and federation agreements developed by MSG-068 (NATO RTO Task Group). These agreements include a set of FOM modules that use and extend existing standard object models, e.g. RPR-FOM and Link 16 BOM.

The basis for the development of the NETN FOM based on this set of modules are national and NATO federations and simulation systems including NATO Live-Virtual and Constructive Federation (NLVC), NATO Training Federation (JTF), Joint Multi Resolution Model (JMRM), German KORA-SIRA (KOSI) federation, Persistant Partner Simulation Network (P2SN) federation, French ALLIANCE Federation, and other simulation systems from Spain, UK, The Netherlands, Australia, Bulgaria, Romania, Turkey, USA and Sweden.

Early in the development of the NETN Reference Federation Agreements it was decided to base the FOM on standards, best-practices and practical experience from the participating nations and organizations. A driving factor in the design of the FOM was to enable a higher level of interoperability between the systems and to allow the use of national simualtion systems in NATO Computer Aided Exercises (CAX).

6.2 Use of FOM modules

The PRP-FOM v2.0 D17 was selected to form the basis for representing Ground Truth for both Platforms and Aggregate Entities. To allow additional information to be exchanged between systems the Platform and Aggregate Object Classes of the RPR-FOM were extended by adding a NETN-Aggregate FOM Module. This module subclasses all the platform object classes and the RPR-FOM AggregateEntity object class.

Another FOM Module called NETN_Logistics was introduced to allow more advanced cross-fedederate logistics operations. It also includes NETN_Facility as a new subclass of BaseEntity. Although RPR-FOM include some basic support for logistics MSG-068 recommendation is to completely replace this with a new way of requesting and providing logistics services. The more general "Service Consumer-Provider" pattern was captured in a separate FOM module and extentions for logistics services was put in another Module. The will allow future modules to utilize the same basic pattern for services whithout including the Logistics module.

The NETN Reference FOM includes the following modules:

- RPR-FOM v2.0 D17
- Link 16 BOM
- NETN Service Consumer-Provider

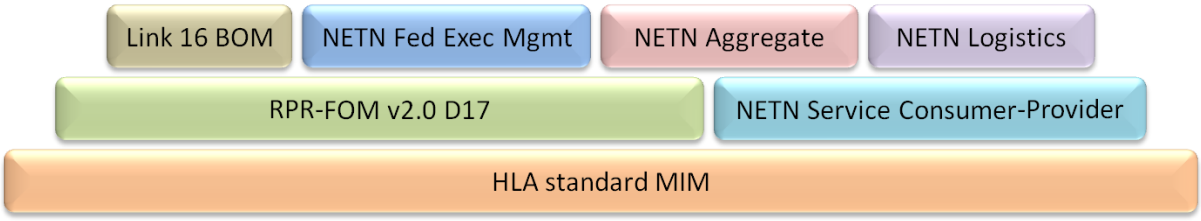


Figure 13 Visualization of dependencies between FOM Modules in the NETN Reference FOM

- NETN Logistics
- NETN Aggregate
- NETN Federation Execution Management

The relationship between the different modules can be seen in Figure 13, a module is dependent on the modules below it in the diagram.

6.3 Logistics FOM module interactions

Figure 14 shows the Interaction class tree for the NETN Logistics FOM module as it is displayed in Visual OMT 2.0. It is easy to see that for example the NETN_SupplyStarted interaction extends an interaction class that is defined in another module, in this case NETN Service Consumer-Provider.

7. Discussion

Some thoughts and challenges that came up during the development of the tools are summarized here.

7.1 Collaboration and the FOM development process

Early in the tool design and experimentation phase we realized that most FOM module editing needs to be focused around a project consisting of several modules. The project typically contains some modules that need to be read-only, from the perspective of a particular federation. An example of this is when several developers share FOM modules under development with each other. When a conflict occurs it is necessary to adjust at least one of the conflicting modules. In some cases the issue to be resolved may be in a read-only FOM, often perceived as "somebody else's" module. In this case it would be a good idea to generate a "change request" from within the tool, instead of updating a read-only module. This is considered for inclusion in later versions of the tool.

On a higher level the entire collaborative process of FOM development may be supported by web based tools. Typical tasks in such an environment would be to handle suggestions and change requests as described in the previous section. This may be a future development.

7.2 Ease of use versus advanced features

It is a challenge to design a tool that both correctly covers the entire standard and, at the same time, is easy to use for beginners. One example of this is how the MIM is handled in Visual OMT 2.0. For the beginner the MIM in itself is not interesting, he probably just want access to the predefined data (such as datatypes). To make it easy for a beginner, the standard MIM is automatically added to all projects and all modules are by default defined as dependent upon it. If ease of use was the only design goal this functionality would be enough, but an advanced user might want to use a non standard MIM, which is also supported by the tool, including the related error-checking.

7.3 Using the OMT glyph as an icon

Visual OMT 2.0 displays the Glyph of the Identification table as an icon in the FOM module overview screen. One issue here is that the Glyph can be any size and any image format. It would be desirable to further standardize the Glyph concept in the Identification table. This would allow programs

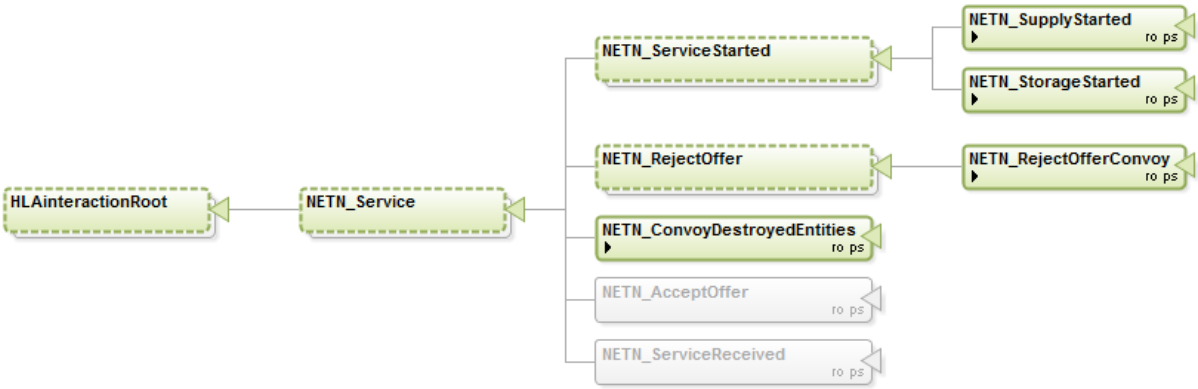


Figure 14: Part of the Interaction class tree for the NETN Logistics FOM

that handle FOMs to use the glyph as an icon for a FOM in their GUI. We recommend standardizing on the GIF, JPG and PNG formats and the size 32 by 32 pixels.

7.4 Attaching Notes to any item

The Notes concept in OMT is difficult to implement in a straight-forward and user friendly way. The OMT format is specified using tables, making it natural to define that it should be possible to place a note on a column. XML is the format used to actually save a FOM file. It lacks the notion of columns. Overall the specified ability to place a note on anything, including placing a note on a note, may result in a complex GUI.

8. Conclusion

The FOM, i.e. the information exchange data model, which is used in a federation is extremely important from both an interoperability and reuse perspective. It is considerably easier to make simulations originally based on similar FOMs to interoperate. It is also more likely that federates can be reused in a federation with a FOM that is similar to their original information exchange data model. HLA Evolved makes it considerably easier to develop and reuse different aspects of a FOM by providing the FOM as composable modules.

Reusable FOM modules ideally should be developed and maintained using a clear and well-defined process. This can be in a top-down process by organizations sharing a common need to support a certain simulation domain. In some practical cases a really useful block of FOM data may be produced in a project and then reused on an ad-hoc basis.

In order to be able to develop FOM modules and to compose entire FOMs from modules, proper tools are needed. They should enable the user to get a clear overview of several combined FOMs, to find errors and to analyze and resolve conflicts between modules. Based on the role of each FOM module in a project the tool should assist a user in keeping standardized modules untouched while developing extensions. A tool should also assist a user in maintaining proper dependencies so that a stand-alone module remains stand-alone and a dependent module only depends on the intended modules. This paper describes how a COTS tool that supports this has been developed.

An example of more general and more specific FOM modules that have been developed in the NATO Snow Leopard federation are also provided.

To summarize, this paper shows how the FOM module concept can take interoperability and reuse to new levels, with the ease-of-use and convenience of a graphical FOM editing tool.

References

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

[2] Vinton Cerf: “Specification of Internet Transmission Control Program”, RFC 675, IETF, www.ietf.org, December 1974

[3] J Postel, J. Reynolds: “File Transfer Protocol (FTP)”, RFC 765, IETF, www.ietf.org, October 1985

[4] R. Fielding et al: “Hypertext Transfer Protocol - HTTP/1.1”, RFC 2616, IETF, www.ietf.org, June 1999

[5] Jonathan B. Postel: “Simple Mail Transfer Protocol”, RFC 821, IETF, www.ietf.org, August 1982

[6] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, To be published.

[7] “TENA - The Test and Training Enabling Architecture, Architecture Reference Document”, https://www.tena-sda.org/public_docmanager/userdocuments/TENA%20ARCHITECTURE%20REFERENCE/TENA%20Architecture%20Reference%20Document%202002.pdf

[8] Thomas Erl: “Service-Oriented Architecture, Concepts, Technology and Design”, Prentice-Hall, July 2005, ISBN 0-13-185858-0

[9] Object Management Group: “Data Distribution Service for Real-time Systems, v1.2”, www.omg.org, January 2007.

[10] MIL-STD-6016B, Tactical Digital Information Link (TADIL) J Message Standard (DRAFT) 15 March 2002

[11] Möller, B and Löfstrand, B, “Getting started with FOM Modules”, Proceedings of 2009 Fall Simulation Interoperability Workshop, 09F-SIW-082, Simulation Interoperability Standards Organization, September 2009.

[12] SISO, “BOM Template Specification”, SISO-STD-003-2006, SISO, www.sisostds.org, 31 March 2006.

[13] Tram Chase, Paul Gustavson, Lawrence M. Root: “From FOMs to BOMs and Back Again”, Proceedings of 2006 Spring Simulation Interoperability Workshop, 06S-SIW-115, Simulation Interoperability Standards Organization, April 2006

[14] SISO, "Standard for: Link16 Simulations", SISO-STD-002-2006, SISO, www.sisostds.org, May 2006

[15] David Hasan: "Using HLA-Evolved Modular Object Models for NASA Constellation Simulation-to-Simulation Interface Specifications", Proceedings of 2010 Spring Simulation Interoperability Workshop, 10S-SIW-012, Simulation Interoperability Standards Organization, April 2010.

[16] Björn Löfstrand, Rachid Khayari, Konradin Keller, Klaus Greiwe, Peter Meyer zu Drewer, Torbjörn Hultén, Andy Bowers, Jean-Pierre Faye. "Logistics FOM Module in Snow Leopard: Recommendations by MSG-068 NATO Education and Training Network Task Group", Proceedings of 2009 Fall Simulation Interoperability Workshop, 09F-SIW-076, Simulation Interoperability Standards Organization, September 2009.

[17] IEEE: "IEEE 1516.3-2003 IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)", www.ieee.org

[18] Joe Sorroche, Riley Rainey: "Directed Energy Modeling and Simulation Experiment Results", Proceedings of 2007 Spring Simulation Interoperability Workshop, 07S-SIW-042, Simulation Interoperability Standards Organization, April 2007.

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.

MARTIN JOHANSSON is Systems Developer at Pitch Technologies and is a major contributor to several commercial HLA products such as Pitch Developer Studio and Pitch Visual OMT 2.0. He studied computer science and technology at Linköping University, Sweden.

BJÖRN LÖFSTRAND is the Manager of Modeling and Simulation Services at Pitch Technologies. He holds an M.Sc. in Computer Science from Linköping Institute of Technology and has been working with HLA federation development and tool support since 1996. Recent work includes developing federation architecture and design patterns for HLA based distributed simulation. He leads the MSG-068 FOM and Federation Design (FAFD) technical subgroup.

ÅSA WIHLBORG is Systems Developer at Pitch Technologies and a major contributor to commercial HLA products such as Pitch Visual OMT 2.0. She studied computer science and technology at Linköping University, Sweden.

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

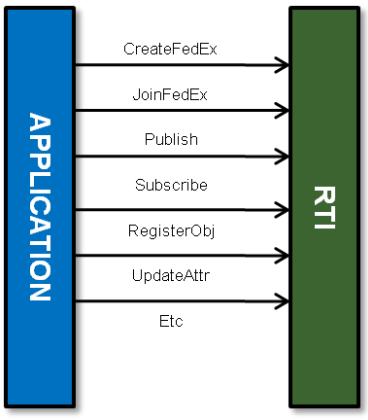


Figure 1: An Application without Middleware

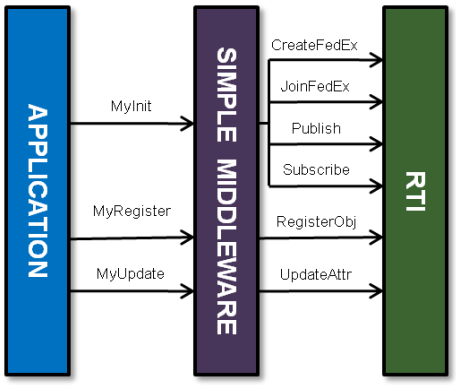


Figure 2: Simple HLA Middleware

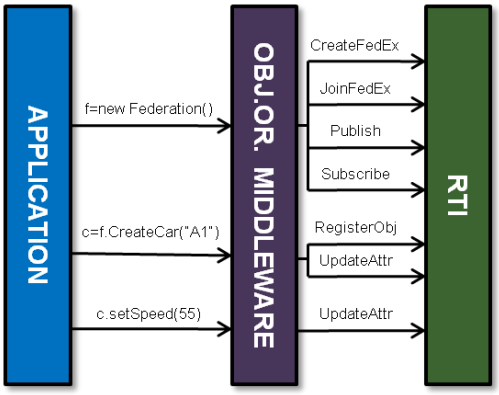


Figure 3: Object Oriented HLA Middleware

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

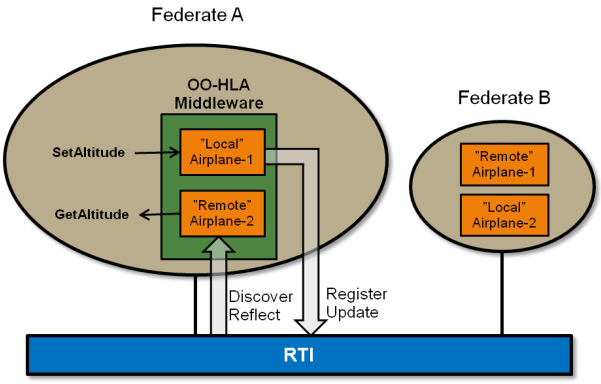


Figure 4: Objects and OO-HLA middleware

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

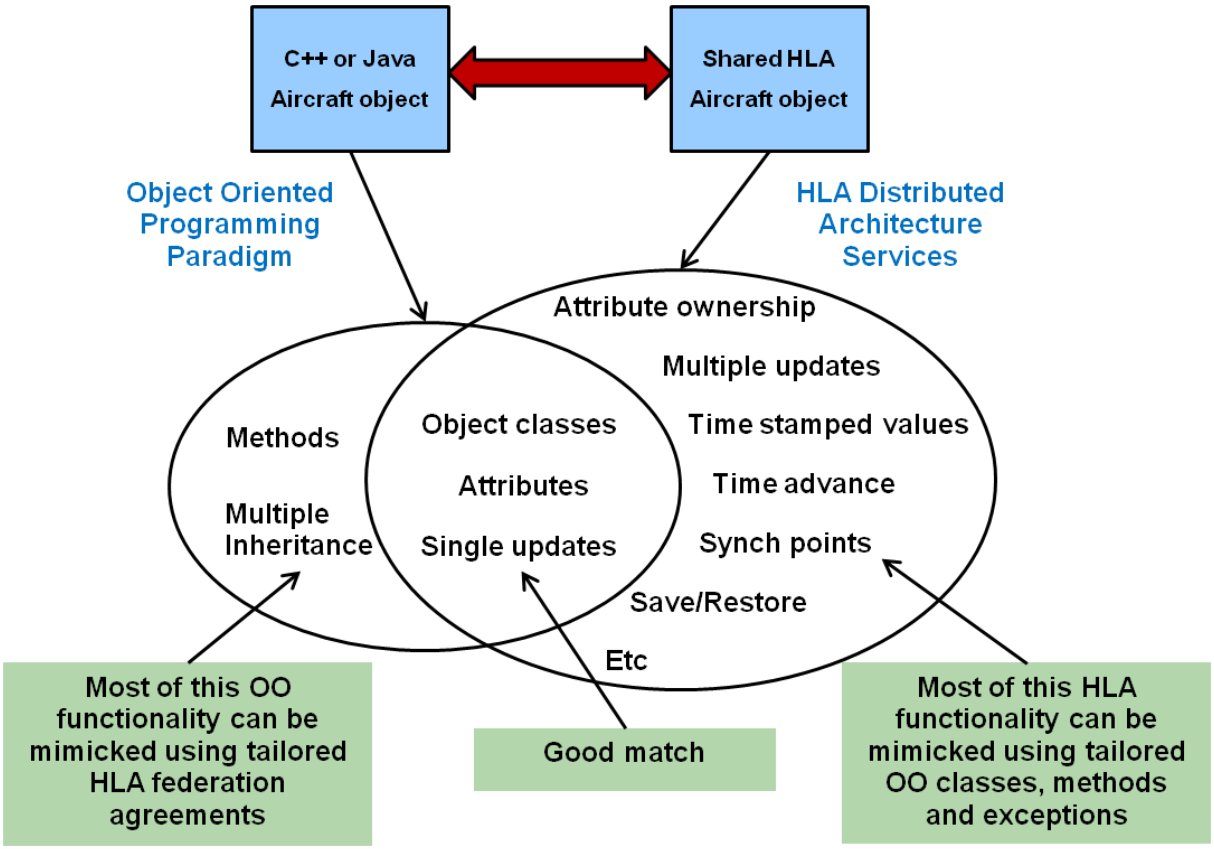


Figure 5: Providing an HLA API using OO and Vice Versa

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 pro-

duction 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

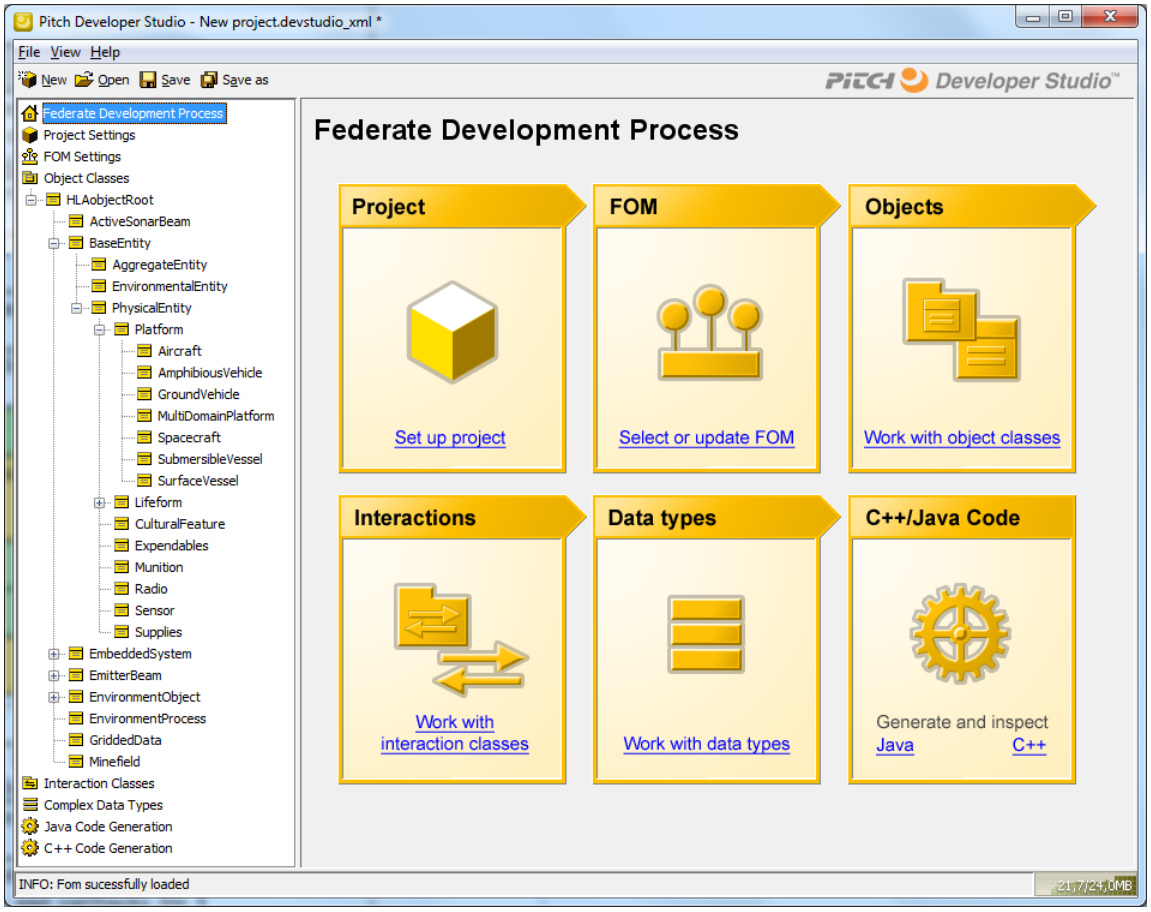


Figure 7: Sample OO-HLA Code Generator Workflow

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:

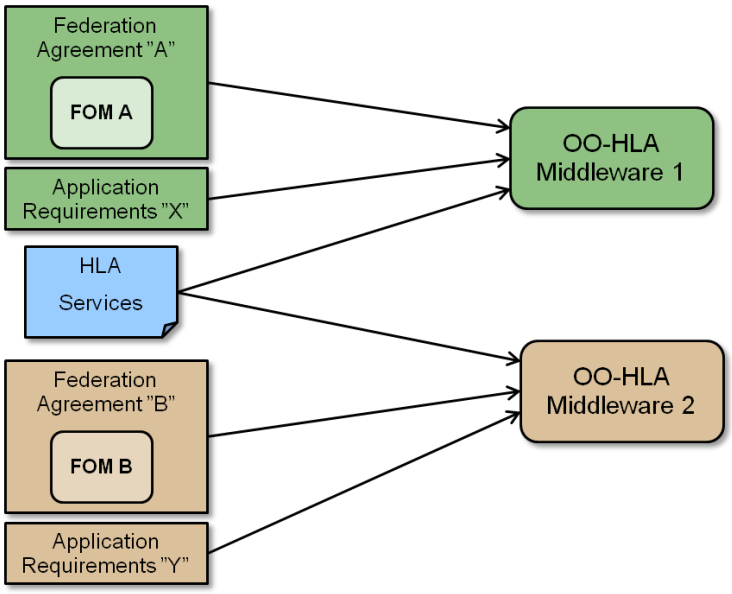


Figure 8: Federation Agreements and Requirements Affects the Optimal OO-HLA Design

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.

Time Representation and Interpretation in Simulation Interoperability – an Overview

Mikael Karlsson
Fredrik Antelius
Björn Möller

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

Keywords:
HLA, Time representation

ABSTRACT: *In the DoD M&S Glossary a simulation is defined as a model operating over time so time is a key element in almost all simulations. Simulations can use time for assigning time stamps to individual data elements as well as for specifying the simulated time for the entire simulator/federate, in particular if the simulation is frame-based. The simulation time needs a binary representation, both in each simulator and in a federation where several simulators interoperate.*

This paper is about how the simulation time is represented and how it is interpreted in HLA Federations and related domains. It mainly focuses on logical or scenario time and doesn't go into detail on sources, usage or correctness of time stamps.

It covers the following:

- *General aspects of binary time representations and in particular time classes used in HLA. An overview of commonly used time representations is given, ranging from the older HLA 1.3 RTI time classes to the standardized time types in HLA Evolved. Issues with floating-point time representations and small time steps are covered in detail.*
- *Specialized time representations such as RPR FOM/DIS time stamps and advanced time classes used for perfect event ordering*
- *Relation to time representations from outside of the simulation domain such as UTC time and other time.*

Finally, this paper identifies some of the typical mistakes that are made in relation to simulation time and some important points and advice for both beginners and advanced developers of federations are given.

1. Introduction

This paper takes a closer look at the representation of time in simulation in general and in simulation interoperability in particular. Special attention is given to the time representations in different versions of HLA, be it fully standardized time representations or commonly used representations.

1.1 Challenges of Discreet Simulation

All software-based simulations are discreet in some sense. The state values, for example the speed of an aircraft, that are calculated and stored are represented in the computer using bits and bytes. These can only have a finite set of values, using for example integers or floating-point representation. In many cases these values represent an approximation of the accurate state value.

The DoD M&S Glossary [1] defines a simulation as a model executed over time. The time representation is also discreet and may in some cases also be an approximation of the intended value. This means that a software based simulation can be seen as discreet both in the state values that are calculated as well as the points in time where these are calculated.

Before taking a closer look at time representations it should be noted that simulations don't necessary

need to be discreet. A mechanical/physical simulator, such as an early pilot trainer, provides a truly continuous simulation over time. It should also be noted that not all software models use models that are calculated over time. An example is damage models where temporal data are sometimes calculated as a function of spatial data.

2 Important Concepts

In order to understand time representations there are several important concepts that need to be understood. This section provides a short introduction with some examples. Most of these concepts are discussed in detail in later sections of this paper.

Wall-clock Time. This is the actual time in the real world when a simulations is executed.

Simulation Time or **Logical Time** this is a representation of physical time within a simulation.

Time Step or **Time Increment**. This is the value by which a simulation repeatedly increments the simulation time. Many virtual simulators (like flight simulators) use a constant time step. These are sometimes referred to as "frame based" since the state in each step is visualized in a frame of a visualization system. The time step may also vary. In event driven simulation the time step will typically be the time to the next known event.

Simulation Executive Loop. The main loop of a simulation program that increments the time value is sometimes known as the Simulation Executive Loop.

Time Resolution or **Precision**. This is the smallest possible difference between two different time values. For an integer the resolution is exactly 1. For a floating-point value the Time Resolution will vary. This concept will be discussed in detail later.

Time Representation. This is the binary representation of the time value, for example a 32 bit little-endian integer or a 64 bit big-endian IEEE-754 [2] floating-point.

Time Interpretation. This is the interpretation of the time value used by the domain model. An integer representation with the value of 47 may be interpreted as 47 seconds or 47 days, depending on the interpretation used.

Time Implementation. This is program code, usually a set of object-oriented classes, which can represent and perform calculation on time values, such as time stamps and time increments. The HLA standard enables the developer to provide his own time implementation classes.

Time Management. HLA defines a set of services for managing the time advance in a federation, as well as the exchange of time-stamped information. An RTI is required to implement these services. These services are described in the HLA standard and will not be covered in detail in this paper.

2.1 Internal vs Shared Time Representations

All members of a federation have to agree on a shared binary representation of simulation time. Each model may have different internal time representations, depending on the model requirements, software technologies and languages used, as well as personal preferences. If the shared representation is different from the simulator's internal representation then it has to provide a conversion between the shared and the internal representation. This conversion has to be accurate enough that any loss of information during the conversion doesn't affect the validity of the simulation. It is not uncommon that information is shared at a lower frequency than the internal model.

3. A Closer Look at Binary Representations

The simulation time needs a binary representation, both in each simulator and in a federation where several simulators interoperate.

3.1 Basic Number Types

There are three basic types of numbers in common use in computers: integer, floating point and fixed point.

The *integer type* represents a mathematical integer. The most common representation is a string of bits, for example a 32-bit integer. The integer type may be signed (capable of representing positive and negative integers) or unsigned (capable of representing only non-negative integers).

The *floating-point type* uses a fixed number of significant digits and is scaled using an exponent. The base for the exponent is usually 2, 10, or 16. A floating-point number can be expressed like this:

$$significant\ digits \times base^{exponent}$$

The following figure shows a floating-point representation of the value 12 648.59 with significant digits 1264859, base 10, exponent 5. The upper row shows the significant digits. The lower row shows the position of the radix point.

```
                          12648 59
12345678901234567890123456789O.1234567890
```

The exponent determines where the significant digits are located in relation to the radix point. Increasing the exponent by one moves the significant digits one step to the left, thereby multiplying the value by 10.

The *fixed-point type* has a fixed number of digits after the radix point. It is essentially an integer with a scaling factor. For example, the number 9.87 can be represented as the integer 987 with a scaling factor of 100. Most computer languages have no built-in support for fixed-point types. The scaling

Type	Description
Integer32	32-bit signed integer
Integer64	64-bit signed integer
Float32	32-bit floating point (24 bits precision, 8 bits exponent)
Float64	64-bit floating point (53 bits precision, 11 bits exponent)

Figure 1: Common binary representations

factor has to be handled manually by the developer when presenting values and when making calculations using values with different scaling factors.

Figure 1 shows the binary representation of common types.

3.2 Interchange Formats

The interchange format for integer and fixed-point types are usually defined as bit strings of a certain length using two's complement.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines interchange formats for floating -point numbers. These formats are encodings (bit strings) that may be used to transfer and communicate floating-point data between systems.

3.3 Precision

Most binary representations have a limited precision. An integer representation has a precision of 1. The typical floating-point representations have a limited number of significant digits. Depending on the value of the exponent, the least significant digit may be 1/4 000 000 or 1/2. This value that the least significant digit (lsb) represents if it is 1 is called the *unit of least precision* (ULP).

It can be said that the fixed-storage representations trade precision for range. When the value goes up, the precision goes down. The figure below shows a decimal representation of the value 12 648.59. The digits are 1264859 and the exponent is 5.

```
                          12648 59
12345678901234567890123455678 90.1234567890
```

The ULP is the value of the least significant digit. In the example above, the ULP is 10^{-2} or 0.01.

If we want to represent a number that is a factor of ten larger then we have to sacrifice precision.

```
                          126485 9
12345678901234567890123455678 90.1234567890
```

The value is 126 485.9 comprised of the same digits as before, 1264859, and the exponent 6. The precision has been reduced, and the value of ULP has increased to 10^{-1} or 0.1. The figure below illustrates how the precision changes as a value increases.

Figure 2: Precision in relation to value

The same rules apply to fixed-storage binary floating-point representations such as float64 although the base is 2 instead of 10. Here's an example with 16 significant digits, 1001011000111010 and the exponent 8. The value is 256 + 32 + 8 + 4 + 1/4 + 1/8 + 1/16 + 1/64 = 300.4531. The ULP is 1/128 or 2^{-7}.

```
                          100101100 0111010
12345678901234567890123455678 90.1234567890
```

3.4 Range

Most binary representations have a limited range. The integer64 representation has a range from $-2^{63} = -9 \times 10^{18}$ to $+2^{63} = 9 \times 10^{18}$. The float64 representation has a range from $(2 - 2^{-52}) \times 2^{1023}$ to 2^{-1074}.
To put these ranges in perspective, the number of seconds in a year is approximately $365 \times 24 \times 60 \times 60 = 31\ 536\ 000$. Thus an integer64 representation can represent a year with a precision better than nanoseconds.
The range of float64 with 1 second as 1.0 and a precision of 1 ms is huge. To represent 1 ms, i.e. 1 / 1 000 accurately, we need at least 11 bits since $2^{-11} = 0.00048828125$. That leaves 42 bits for seconds which can represent almost 140 000 years.

3.5 Epsilon

Epsilon is the name of the smallest value that can be represented in a particular binary representation. The following table shows the epsilon for different binary representations.

Representation	Epsilon
integer32	1
integer64	1
float32	2^{-149}
float64	2^{-1074}

Figure 3: Epsilon for common representations

In the integer representations, the epsilon value and ULP remain equal and constant regardless of the value being represented. The floating-point representations on the other hand have a ULP that becomes larger as the value gets larger.

4. Floating-Point Issues

The use of floating-point calculations in computers have a number of non-intuitive behaviors.

4.1 Exact Values

Binary representations are unable to make an exact representation of some values. For example, a decimal representation can only represent the value 1/3 as an approximation. This is quite intuitive to humans as we have the same problem in the common decimal system. The value 1/3 generates an infinite number of decimals, 0.33333333..... Less intuitive is the fact that float64 cannot represent the value 0.1 exactly. In the same way as the value 1/3 generates an infinite decimal representation, the value 0.1 generates an infinite binary representation.

Most developers would hesitate to use the value 1/3 as time step since it cannot be exactly represented in a decimal form. However, they will use the value 1/10 (0.1) as time step with a float64 representation without realizing that this value cannot be exactly represented in a base 2 format.

4.2 Confusing Arithmetic

Adding large and small values using floating-point representations can yield unexpected results since the smaller value can fall below the ULP of the large value and therefore not affect the large value. See illustration below.

```
                          12648 59
+                           0 0001
12345678901234567890123455678 90.1234567890
=                         12648 59
```

A special case of this is that Epsilon may be smaller than the ULP since ULP may change depending on the value. This means that taking a number and adding epsilon to it is not guaranteed to yield a new number.

4.3 Accumulated Errors

What's 0.1 × 10? Is it the same as 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1? You'd think so, but it actually isn't when using floating-point arithmetic. Here's a simple Java program that performs the calculation.

```
public class TestArithmetic {
   public static void main(String[] args)
   {
      double step = 0.1;
      double multiplied = step * 10.0;
      double accumulated = 0;
      for (int i = 0; i < 10; i++) {
         accumulated += step;
      }
      System.out.println(
         "multiplied = " + multiplied);
      System.out.println(
         "accumulated = " + accumulated);
   }
}
```

The output of this little program is:

```
multiplied = 1.0
accumulated = 0.9999999999999999
```

This discrepancy may cause problems, for example if a federate is planning to perform an action at time 1.0 and is checking the current time to see if it is equal to that value. The values will never be equal and the action will never be performed. It is also very likely that the current time is presented as a rounded value which will be 1.0, thereby hiding the problem and making debugging even more complicated.

4.4 Comparing for Equality

Due to the limited precision of floating-point numbers, comparing for equality is unlikely to give the intended result.

```
if (result == expectedValue)
```

A common attempt to solve this problem is to say that the values are considered equal if they are closer than a predefined distance, say 0.00001.

```
if (abs(result - expectedValue) < 0.00001)
```

This predefined distance may work fine during development of a simulation where simulation time never exceeds 1 000, but when the simulation is put into production and it runs for 100 000 steps the predefined distance 0.00001 falls below the precision of the time representation and the comparison fails.

4.5 Changing Behavior

In some situations, the result of floating-point calculations can depend on such unexpected factors as compiler flags. For example, the following C program prints one value when compiled in optimized mode, while it prints another value when compiled in standard mode.

```
double foo(double v) {
   double y = v * v;
   return (y / v);
}

main() { printf("%g\n", foo(1E308));}
```

The reason for this behavior is that the computation v × v is done in extended precision, with a larger exponent range. In optimized mode, the extended precision result of v × v stored in a register is used in the calculation of y / v. In standard mode, the result of v × v is stored in the double precision variable y as positive infinity due to the overflow. The variable y is then loaded and used in the calculation of y / v to yield positive infinity.

HLA Version	Name	Binary Representation	Comment
HLA 1.3	LogicalTimeDouble (“NG”)	Integer64	Integer value to be divided by one million
HLA 1.3	LogicalTimeDouble (“DMSO”)	Float64	Not Epsilon safe
HLA 1516 (2000 API)	LogicalTimeDouble	Float64	Not Epsilon safe
HLA 1516 (2000 API)	LogicalTimeDouble (alternate)	Integer64	Integer value to be divided by one million
HLA 1516 (DLC API)	LogicalTimeImpl	Float64	Not Epsilon safe
HLA 1516 (DLC API)	LogicalTimeImplFloat	Float64	Not Epsilon safe
HLA 1516 (DLC API)	LogicalTimeImplInteger	Integer64	
HLA Evolved	HLAfloat64Time	Float64	Standardized. Epsilon safe
HLA Evolved	HLAinteger64Time	Integer64	Standardized.

Figure 4: Common and standardized time classes in HLA

5. Time Implementation in HLA

Throughout the lifetime of the HLA standard, different time representations have been used. Here's a historic overview, including time representations in related standards. Figure 4 provides a list of time representations from the HLA 1.3 [3] standard through HLA 1516-2000 [4] and HLA Evolved (IEEE 1516-2010) [5].

5.1 Custom Time Classes

Ever since the first version of HLA, custom time classes have been supported. In fact, the standard hasn't provided any standardized time classes until HLA Evolved. The time types LogicalTimeDouble and LogicalTimeFloat in HLA 1.3 and HLA 1516 have simply been example classes provided by RTI developers. These classes have then become de facto standards.

5.2 Common and Standardized Time Classes

Figure 4 shows the common time classes usually found in HLA 1.3 and HLA 1516-2000 RTIs. Note that none of these common time classes that use floating-point representations are epsilon safe.

HLA Evolved (IEEE 1516-2010) finally introduces two standardized time representations; HLAinteger64Time and HLAfloat64Time, where the names, representations, behavior and even the serialized form have been standardized so they are guaranteed to be compatible between RTIs and RTI vendors. HLA Evolved, like the previous HLA versions, also allows the use of custom time classes.

5.3 Compatibility

The API for time classes has changed between different HLA versions, which means that it is not possible to use a time representation from one HLA version with another HLA version.

The implementation of the time representation has to be available to all federates in the federation. So if the federates are implemented in different environments (e.g. operating system, compiler version, programming language) the time implementation has to be available in all the different environments.

5.4 HLA Evolved and the Vanishing Epsilon

The fact that epsilon can fall below ULP disrupt the internal calculations made in an RTI, for example to guarantee that all messages have been delivered before allowing a federate to advance to a certain point in simulation time.

In HLA Evolved, the time implementations are required to have special handling of the vanishing epsilon. The rule, in a simplified form, is that taking a number and adding a non-zero number to it shall always yield a different number. This requirement doesn't completely fix the problem with vanishing epsilon, but it will prevent the federation from grinding to a halt.

6. Some Other Time Representations

This section provides a overview of other, related time representations. It also gives an example of a more advanced time class.

6.1 UTC Time

UTC [6] time or Coordinated Universal Time is used by many computer application that need a universal time. UTC time is not affected by daylight saving time and it is also the “common” time zone that all other time zones relates to. UTC in essence is Greenwich Mean Time (GMT). UTC is

Type	Unit	Required Time Resolution	Resulting Time Range
integer32	second	1 second	2^{31} / 86 400 / 365 = 68.096 years
integer32	millisecond	1 millisecond	2^{31} / 1 000 / 86 400 = 24.855 days
integer32	microsecond	1 microsecond	2^{31} / 1 000 000 / 60 = 35.791 minutes
integer64	microsecond	1 microsecond	2^{63} / 1 000 000 / 86 400 / 365 = 292 471.209 years
integer64	nanosecond	1 nanosecond	2^{63} / 1 000 000 000 / 86 400 / 365 = 292.471 years
float32	second	1 second	2^{24} / 86 400 = 194.181 days
float32	second	1 millisecond (11 bits)	2^{13} / 60 = 136.533 minutes
float32	second	1 microsecond (20 bits)	2^{4} = 16 seconds
float64	second	1 millisecond (11 bits)	2^{42} / 86 400 / 365 = 139 461.14 years
float64	second	1 microsecond (20 bits)	2^{33} / 86 400 / 365 = 272.385 years
float64	second	1 nanosecond (30 bits)	2^{23} / 86 400 = 97.090 days

Figure 5: Precision and range for different (signed) data types

specified using a date and a time value.

6.2 Time in DIS

The DIS [7] PDU header contains a *timestamp* when the data was generated. This is either an *absolute timestamp* using UTC time or a *relative timestamp* relative to an arbitrary starting point. Relative timestamps are used when the clocks of the simulation applications are not synchronized.

The timestamp uses an unsigned 32-bit integer representation. The least significant bit is used to indicate absolute or relative timestamp and the remaining 31 bits is used to represent time passed since the beginning of the current hour. Each time unit represents 3600 s / (2^{31} - 1) = 1.676 µs.

6.3 Time in GRIM-RPR

The GRIM-RPR [8] in HLA uses the DIS timestamp in the user-supplied tag for some of the HLA services. The first 8 bytes of the user-supplied tag contains the 32-bit DIS representation encoded as an ASCII string (without the usual terminating '\0') using hexadecimal ASCII characters (0-9 and A-F), including leading zeros.

6.4 Tie-breaking Time Implementation

An important property of some simulations are repeatability and independence on the number of nodes used during the execution. Repeatability problems can occur when multiple events are scheduled at the same time. [9] To achieve repeatable results, special care must be taken to ensure that the events are processed in the same order every time the simulation is executed. One approach is that the model developer adds or subtracts tiny epsilon increments to event times during event scheduling to manually produce unique ordering. This type of solution usually breaks down quickly as models get more complex. In addition, this practice violates the modeling since events will not occur at their calculated time.

A better way to guarantee repeatability is a time implementation that provides unique timestamps for all events by adding special tie-breaker fields. The WarpIV Kernel [10], a high-performance computing framework, provides a time implementation with these characteristics.

7. The Time Agility Challenge

HLA doesn't have a predefined information exchange model for any particular domain. Instead a Federation Object Model is used. This is an XML document that describes the shared objects and attributes (like aircrafts) and shared interactions (like start/stop commands or munition fire). Starting from HLA 1516-2000 the FOM defines the exact representation of the data, such as integer size, byte ordering, record layout, etc. A federate needs to format (encode) an attribute or parameter value according to the FOM before sending it to other federates. When it receives data from another federate it needs to decode the data.

One advantage of HLA is that the FOM concept makes it possible to implement general tools that support any FOM. The tool reads the FOM and can then use this information to dynamically decode the data that is received, for example for display purposes.

It is a bigger challenge for an HLA tool to be flexible with respect to the time representation. Time values are received as byte arrays. First this value needs to be decoded, according to the time repre-

sentation table in the FOM. Secondly, an HLA tool that needs to support Time Management, needs to match this to the corresponding time implementation in the code.

The challenge here is that there are numerous ways to represent and interpret time. Time implementation classes (code) need to be provided for each of them in order to be able to perform time calculations.

To reduce this problem HLA Evolved introduces two distinct, standardized time representations: HLAinteger64Time and HLAfloat64Time. If simulation developers adopt these representations it will reduce the effort to make simulations interoperable. It will also simplify the development of time agile tools.

8. Advice on the Choice of a Time Representation

The following is a simple list of advice when choosing a time representation:

a) Understand the limitations of the time representation that you intend to use. Integer representation will have their obvious limitations in resolution and highest possible value. For floating-point representations, big values may result in two different problems. If the time implementation doesn't guarantee an increment then simulation time will effectively stop. If an increment is guaranteed, then time steps may be too large, unintentionally "speeding up" the simulation. Each time representation has a "sweet spot" in its effective range.

b) Use time representations that have a big enough value range for your simulation. If your simulation starts at a large time value, consider offsetting this so that is starts at zero. Remember that large values may have larger ULP. Figure 5 shows the range for a number of types given a specific precision.

c) Consider using integer or fixed-point time representations. These are considerably easier to understand and debug. If a floating-point value is required in the calculation this can be achieved in the interpretation of the value. You may for example consider using an integer that is interpreted as microseconds.

d) If you use floating-point time representations, make sure that it provides safe increments of time. You should still try to stay in the range of time values where this feature isn't necessary.

e) Use the standardized time types according to HLA Evolved (IEEE 1516-2010): HLAinteger64Time and HLAfloat64Time, even if you are developing for an older version of the standard. This allows you to minimize risk and take advantage of the time implementations provided by the RTIs. In some cases RTIs also provide compatible implementations for the older APIs.

f) Consider writing code that can detect where the time values leave the "safe range" and warn the user when the time of his scenario goes outside of this range.

9. Conclusions

This paper has provided an introduction to several aspects of time representations. The challenges of the vanishing epsilon has been introduced. Some challenges on creating time agile tools have also been presented. Some advice on how to choose a time representation has been given.

One important long-term solution is to strive towards more standardized time representations, two of which are provided in HLA Evolved (IEEE 1516 -2010).

No matter which time representation that is chosen, it will have limitations and the simulation developer needs to understand the nature of these limitations.

References

[1] "DoD M&S Glossary (5000.59-M)", MSCO, www.msco.mil.

[2] IEEE: "IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic", www.ieee.org.

[3] "High Level Architecture Version 1.3", MSCO, www.msco.mil.

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

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

[6] "Coordinated Universal Time", http://en.wikipedia.org/wiki/Coordinated_Universal_Time.

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

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

[9] Steinman, J., "Introduction to Parallel and Distributed Force Modeling and Simulation", 09S-SIW-021, www.sisostds.org.

[10] "WarpIV Kernel", http://www.warpiv.com

Author Biographies

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 infra-

structures 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.

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 Support Group.

Developing an HLA Tutorial: Philosophy and Best Practices

Björn Möller, Pitch Technologies, Sweden
Steve Eriksson, Pitch Technologies, Sweden
Åsa Wihlborg, Pitch Technologies, Sweden

bjorn.moller@pitch.se
steve.eriksson@pitch.se
asa.wihlborg@pitch.se

Keywords:
HLA, Tutorial

ABSTRACT: *A standard, like HLA, needs to be exact, complete and unambiguous. This may not be optimal for a beginner wanting to learn HLA. An up-to-date HLA Tutorial document has now been developed. This paper summarizes some of the philosophy of the tutorial. Several best practices on how to use the standard are also covered in the tutorial.*

One of the philosophies of the tutorial is to describe how services from different service groups are used to solve common tasks instead of strictly describing the structure of the HLA standard. Another philosophy is to emphasize how concepts from the HLA Interface Specification and the HLA Object Model Template are used together.

The best practices covered include low-level aspects like optimal memory allocation, handling of HLA service exceptions as well as life-cycle management of shared objects. Selected architectural aspects are also covered, like the use of a federate architecture that separates HLA concerns from domain simulation concerns, federate testing strategies and a basic Federation Agreement sample.

The tutorial, together with C++ and Java sample code, is freely available to industry, academia and anyone interested in learning HLA.

1. Introduction

A publicly available HLA Tutorial [1] has been developed. This paper describes the tutorial and summarizes the philosophy of the tutorial, the best practices presented and finally some thoughts and insights gained during the development.

1.1 Standards versus Tutorials

A beginner wanting to learn HLA, or any standard, may initially be disappointed by the typical standards document. Every feature is described in the utmost detail with a highly specialized terminology. It may take quite a lot of reading to figure out which combination of services that are needed to solve a particular problem. However, the primary purpose of a standards document is to provide a complete, consistent and unambiguous specification, not to give an introduction for a beginner.

Teaching HLA is more similar to telling a story, introducing new concepts, step by step, as they are necessary to fill a particular purpose. Every service should be put into a context of how it is used. The user needs to understand the main principles and central parts of the standard. Additional details can be studied later on when needed.

1.2 Evolving standards and best practices

HLA was incepted in the mid 90's. HLA 1.3 [2] was released in 1998, HLA 1516-2000 [3] in 2000, the SISO DLC [4] standard in 2004 and HLA Evolved [5] in 2010.

There is very little introductory material aimed at developers. The original HLA book [6] was released in 2000 and is based on HLA 1.3. There is also an older programmers guide [7] that came with the DMSO RTI that is also based on HLA 1.3. There are some practical migration guides for migrating to HLA 1516-2000 [8] and HLA Evolved [9]. But in general there is a lack of a practical and up-to-date tutorial for HLA. This may not be a problem for the seasoned HLA developer, but it is a barrier to entry for new persons and organization that need to start using HLA, thus limiting the success of HLA.

Not only the standard itself but also best practices for its usage evolve over time, based on experiences from building federations. Many projects and organizations have reached consensus on how to best use the HLA architecture and services over time. This also needs to be reflected in a tutorial of 2012.

1.3 The HLA Tutorial – why and how

The HLA Tutorial was developed to promote the HLA standard and to make it easily accessible for industry, academia and anyone interesting in the subject. Over the past years there has also been a growing user base, in particular in the civilian domain, that has requested an easily available, up-to-date tutorial. Another issue is that there is currently no tutorial that focuses on how federation agreements, object modeling and HLA services play together.

The tutorial is intended to complement the standard. The tutorial document and the samples represent many man-months of work from experienced HLA instructors and developers. The tutorial is intended to be vendor neutral (although screen shots are generally taken from Pitch products). The source code should work with any HLA Evolved RTI.

The HLA Tutorial builds upon almost 15 years of experience from teaching HLA in courses and seminars in more than a dozen of countries in four continents.

2. Overview of the Tutorial

The tutorial is freely available as a PDF document that may be redistributed. There is also a bigger package, the "HLA Evolved Starter Kit", which contains sample FOMs and federates in C++ and Java as shown in Figure 1. The tutorial can be downloaded separately or as a part of the Starter Kit.

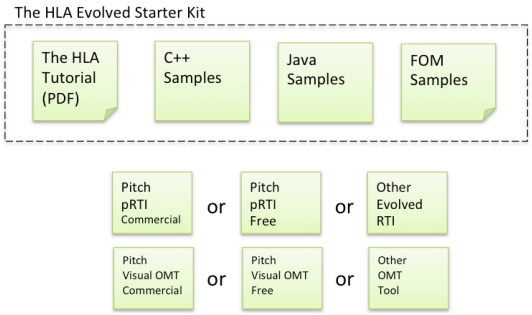


Figure 1: Components of the HLA Evolved Starter Kit and additional HLA software

These samples are intended to work with any HLA Evolved compliant RTI. In case the user does not already have an HLA Evolved RTI or an HLA OMT tool, free versions are available for download.

2.1 Structure of the tutorial

The tutorial consists of two parts but, to date, only the first part has been released. Part one focuses FOM development and the basic HLA services: federation, declaration and object management. Part two focuses on Ownership Management, Time Management, DDM, MOM and other more advanced concepts. Part one contains the following major sections:

Chapter 1 gives an introduction and describes the purpose of HLA (interoperability and reuse), a number of practical applications of HLA and a few words on policy, standardization and how standards can enable a market place.

Chapter 2-3 describes the architecture, topology and services of HLA, provides an overview of the architectural aspects of HLA and introduces the basic terminology without going into detail. It presents HLA as a modern, service oriented, architecture based on a service bus.

Chapter 4-9 provides a step-by-step description of how to build federates for a Fuel Economy Federation. The FOM development and the use of HLA services in the federate code are intertwined to illustrate how they play together. The PDF document contains simplified pseudo-code whereas the C++ and Java samples provide more details such as exception handling.

Chapter 10 describes basic and intermediate techniques for testing and debugging federates and federations.

Chapter 11 describes different types of Object Oriented HLA. It also provides a small amount of sample code.

Chapter 12 summarizes the tutorial, provides a short description of DSEEP and points at some additional HLA services that will be covered in part 2 of the tutorial, for example Time, Ownership and Data Distribution Management as well as the Management Object Model.

Appendix A and B provides a complete Federation Agreement and FOM for the Fuel Economy Federation.

Appendix C provides federate descriptions and file formats (scenario file and federate configuration file) for the Fuel Economy Federation.

Appendix D provides eight lab instructions for readers that have installed the sample federates on his computer. The instructions describes how to run the federates and what code to inspect for each chapter. The advanced user may also modify or extend these federates.

Appendix E describes the classical Restaurant federation and how to run it.

Appendix F gives an overview of the HLA Rules.

2.2 The Fuel Economy Federation

The main example used in the tutorial is the Fuel Economy Federation. Its purpose is to evaluate the

fuel consumptions when multiple cars, simulated by different federates, drive along a particular route specified in a scenario file. The interoperability aspects are covered in detail, but the scenario format and fuel consumption models are intentionally simplified.

The federation has three types of federates. The **CarSim** simulates one or more cars. The **MapViewer** visualizes the cars on a map and in a list. The **Master** controls the federation. The federate types matches what is found in many real federations.

In addition to exchanging information about cars and fuel there are two patterns in the federation. The Master controls the scenario selection. Participating federates signal whether they were able to load the selected scenario or not. The Master also controls the start and stop of the scenario time, i.e. the simulation. The scenario is run at scaled real time, for example 5.5 or 1.0 times the real time. These patterns are described in the included Federation Agreement.

2.3 The Restaurant Federation

This federation is provided for the advanced reader. It models a sushi restaurant where chefs prepare sushi and places them on boats that transport them to customers. It was originally developed for HLA 1.3, using an older version of Java. It was originally included in the HLA book [6]. It has now been migrated to HLA Evolved. Some of the code does not always follow what we today consider best practice. This sample is included despite this since it uses a wide range of HLA services.

3. Philosophy

There are many ways to teach HLA. This section summarizes both lessons learned from giving a large number of courses as well as some design choices. Here are the most important philosophies.

A tutorial aimed at the practitioner

HLA and interoperability may well be described from a theoretical or architectural point of view. In this case we have decided to target practitioners that need to develop real HLA federates and federations. This requires more practical how-to advice, and reasonably complete code examples.

Simplify and focus on what´s useful

Many of the chapters have been shortened and simplified several times. More than half of the text of the first draft has been removed. This text was mainly advanced discussions. The text has been simplified as far as possible to make it easy to grasp. New concepts have been introduced only when they can be easily understood.

Present HLA as a modern software architecture

Ten years ago HLA was often presented primarily as a US DoD strategy. While HLA still has a strategic position in NATO and the US DoD, todays developers are more interested in its virtues as a modern software architecture, in particular if they work outside of the defense domain. There are today several related architectural styles that can help people understand HLA, for example Information Bus and Enterprise Service Bus.

Learn by common tasks

It may be tempting to structure an HLA tutorial according to the three standards documents and their chapters. One problem with this is that it takes a long time before the relationships between different parts of the standard starts making sense. The philosophy chosen is instead to show how to perform common tasks. Figure 2 shows an example of this. It illustrates which part of HLA that you need to use to send an interaction.

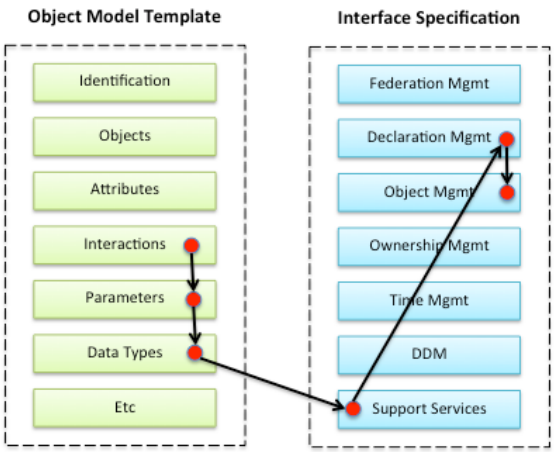


Figure 2: Parts of the HLA specification used to send an interaction

First you need to design an interaction with parameters in the OMT. You also need to define data types for any parameters. Then you need to use the services in the Interface Specification to get handles, publish and subscribe the interaction and finally send it. You may also consider using encoding helpers, which is not shown in figure 2.

The task-oriented approach clarifies how different parts of the standard are used together. It also provides more "instant gratification" to the developer.

Highlight federation agreements and object modeling

Many federate developers today underestimate the value of the FOM concept and the importance of quality FOMs. The approach described in Figure 2

also helps the developer understanding the relationship between the FOM and the federate code.

The FOM, on the other hand, only provides a lower level description of the overarching Federation Agreement, which gives the bigger picture of how to use the information in the FOM.

While other efforts, like FEAT [10] attempts to advance the state of the art with respect to more advanced federation agreement, this tutorial seeks to introduce newcomers to a very simplistic but correct and complete federation agreement.

Use simple pseudo-code in the main text

To clarify how the services are used, the main text of the tutorial uses simplified pseudo-code. The C++ and Java sample code, on the other hand, contains full detail and extensive exception handling. Since many beginners tend to reuse sample code it is important to avoid simplifying for example exception handling. Inferior exception handling often makes federation debugging more time-consuming than necessary.

Use simple english

It is likely that the majority of todays´ HLA users do not have English as their first language. Because of this the tutorial has been written in a simple and friendly tone.

4. Best Practices

HLA offers a large number of services. The HLA tutorial presents how to use them in a simple federation, step by step. In many cases there are several ways to approach a problem, to design a federate and to use the HLA services. The HLA tutorial tries to present a number of best practices that are on an appropriate level for beginners. This section presents some of them.

Note that more advanced or large federates and federation may need to deviate from some of these best practices. Still it is highly useful for a beginners to get their first federates well designed from a general perspective.

4.1 Architecture and design

These best practices apply to the overall architecture and design of federates and federations.

1. Carefully design your federation agreement and FOM before attempting to write any code. If the federation agreement already exists, study it carefully. Starting to design your federates without a proper federation agreement and an agreed upon FOM often leads to costly redesign of the federates. When designing other software it is often possible to "wing it" when problems occur, but since federates that you write need to be compatible with other federates it is not as easy in a federation. Not having a clear view of how everything is supposed to work when starting can lead to different behaviors in different federates depending on the implementers interpretations.
2. Use a coordinated approach for handling scenario time and scenario data across the federation.

   Even if your federation is not time managed the concept of scenario time will exist in the simulators. In many cases federates in a federation will execute in scaled real time. Think through how time will be handled and use a similar approach in all federates when possible. If one federates supports pausing but none of the other federates does, then that feature will not be useful in the federation. Running at a faster or slower pace, pausing and jumping in time, are some things that you should consider even for non time-managed federations.

   Using common scenario data removes the possibility that simulators disagree on the content of the simulation. It also minimizes scenario development work and minimizes the risk for uncorrelated scenario data.
3. Use a specialized federate for starting, stopping and selecting scenario. This follows the principle of separation of concerns. The simulators are experts in simulating a system or parts of a system, not coordinating simulators in a distributed environment. Centralized handling and coordination makes it easier to have clear and well-known states of the simulators in the federation execution.
4. Put the HLA interface code in a separate module. This is also a question of separation of concerns. In software development it is good practice to create modular and loosely coupled system. Encapsulate changeable design decisions. The different modules should have a specific purpose and be as independent as possible from other modules. This approach makes it easier to understand and develop the different modules. It also makes it easier to find faults in the system and changes are easier to make since they won't affect the whole system. Another advantage of using a modular design is that simulation experts can develop the simulation part and

HLA experts, maybe external developers, can develop the HLA part.

5. Tailor the HLA interface module to the specific subscription needs of each federate for best performance. It is often a good idea to create general and reusable software components. However, when building an HLA module for a certain simulator and federation it might not be the best choice. The major problem is that it may subscribe to more information than needed in a particular federate, causing increased usage of CPU, memory and networking resources. It might also make your system more error prone, harder to understand and harder to maintain. Our recommendation is to optimize for the current environment. Modular design makes it relatively easy to change or switch the tailored HLA interface.

4.2 Program structure

These best practices apply to the use of HLA services within a federate.

1. Register objects without reserving HLA object instance names. Instead use an attribute for naming. HLA object instance names are globally unique. It's relatively costly to register a unique name in the federation. The central RTI component has to check for uniqueness and your application has to handle any error thrown by the RTI should the name already be taken. Instead, have the RTI create a unique name and use an attribute in the object for the name.

2. If you still do reserve names and you need to reserve more than a few HLA object names, use the Reserve Multiple Object Instance Names service.

3. Use a table or hash map for storing the references for discovered object instances. If possible consider a lookup function to quickly locate instances based on name, handle and other relevant keys. Consider having one table or hash map for each object class. One of the functions an HLA module is likely to execute very often is to translate between a simulation objects id and the corresponding HLA object.

4. Implement the Provide Attribute Value Update callback so that other federates, possibly late joiners, can get the mot recent values. Use this in conjunction with the Auto Provide switch, at least for smaller federations. Failing to implement this support can make it impossible for other federates to join the federation when it is already executing. Not supporting late joiners leads to problems even for federations with a fixed federate lineup. For example, if a federate crashes during an execution it cannot rejoin the federation. This might make it necessary to restart the whole federation every time a federate crashes.

4.3 Low level programming

These best practices apply to detailed programming aspects. In most cases they are independent of the programming language used.

1. Allocate memory for objects like encoding helpers in the initial part of the program, not in the main loop. Encoding helpers can be relatively expensive to create and they are likely to be used very often. For optimum performance you should therefore create them once during initialization.

2. Get handles from the RTI in the initial part of the program, not in the main loop.

3. Handle all HLA service exceptions. Instead of just terminating your application if it encounters an exception you should handle it so the application degrades gracefully or at least clean up before terminating. Clean up includes resigning and disconnecting from the federation and perhaps hand over ownership of simulation objects.

4. Be careful with exceptions that are related to any user input or the current technical environment (for example cannot find FOM file, bad IP address). Give the user an opportunity to fix these. Configuration problems related to federation name, FOM file to use and IP address of the RTI central component can easily be remedied by the user or a technician so make sure to give clear error messages.

5. Use encoding helpers to get correct encoding. Utilizing the provided encoder classes is also a good way to be sure that the data is encoded correctly for any operating system, CPU and development environment.

6. Assume that all data received may be incorrect or incorrectly encoded. As with all application development, don't make your code rely upon external components to provide you with correctly formatted data. Failing to do this may open up for security breaches and unnecessary termination of your application.

7. Several callbacks (like Reflect Attribute Values) have several different

implementations, with different parameter sets, in the API. Handle al versions, for example by dispatching them to one common implementation. In most cases it's a good practice to handle update of attributes in a uniform way. Having one implementation for each overloaded version of the callback method makes your code harder to maintain. The more code you write the risk of introducing bugs increase. Subtle differences in the overloaded methods may create hard to find bugs. More code often means more tests and maintenance.

4.4 Testing

These best practices apply to the testing of federates and federations.

1. Verify that all operations, like declarations and object registrations, work as intended by inspecting the RTI user interface.
2. Verify that your federate sends correctly encoded data before trying to use it in a federation. A well behaved federate should only send properly formatted data according to the FOM. Don't assume that the other federates can handle bad data otherwise you may make them terminate ungracefully.
3. Test your federates against known-good federates and recorded data. This is an excellent way to pinpoint where in a federation a problem occurs. For example if your simulated entities don't show up where they are expected in a viewer you can connect it to another proven viewer to determine if it's your federate that sends bad position data or the viewer that displays it incorrectly.

5. Discussion

As can be understood from the previous descriptions a lot of topics have been covered in the tutorial. In this discussion we would like to focus on some topics that we found more difficult to fit into the tutorial.

5.1 The SOM

While the SOM is in no way a difficult concept to explain, it was difficult to find an obvious place to describe how it fits into the federation development process in the tutorial. One approach used to present the SOM is the use of Publish/Subscribe Matrix. Figure 3 shows such a matrix. Each row represents an attribute or an interaction. Each column represents a federate. In each cell we then specify whether the federate publishes or subscribes to that attribute or interaction.

	Fed A	Fed B	Fed C
Car.Name	Pub	Pub/Sub	Sub
Car.Position	Pub	Pub/Sub	-
Start	Pub	Sub	-

Figure 3: A Publish/Subscribe Matrix

This type of matrix offers a good starting point for a discussion about SOMs.

5.2 The HLA Rules

In most training events we have found that the HLA Rules seem very abstract when presented early in an HLA training event. After a few days of HLA training most participants find them very easy to understand. Several rules are based more or less on common sense, saying that a federate shall hold what it promised to do. Other rules are less trivial and may require some additional discussion, for example that the RTI will only transmit, not store, any data values for attributes and interactions. Several of the rules relate to the SOM, which is not covered in detail in the tutorial. The approach in the the tutorial is to put the HLA Rules in an appendix.

5.3 Getting hold of the HLA specification

The HLA Tutorial makes extensive references to the HLA specification, which is not available for free. Some universities have purchased full access to all IEEE standards, which makes it easy for students and staff to get them. Other readers may be SISO members, which gives them full access to the standard at a modest price. Still a large number of readers will have difficulties getting the HLA specification for various reasons. Students in many parts of the world may consider the price high. Complicated administration may slow down a purchase for a reader in a large organization. Today many readers expect software standards to be freely downloadable from the Internet, which may create some disappointment.

5.4 Understanding commonly used FOMs

Most people learning HLA are interested in understanding not only HLA but also commonly used FOMs. For the defense sector this is usually the Real-time Platform Reference FOM (RPR FOM) [11]. This is a fairly advanced FOM with complex data types. One challenge is that such a FOM is too complicated to be used for explaining basic HLA concepts. It may actually require a tutorial on its own. Another challenge is that many new HLA users come from other domains than defense.

For the HLA tutorial we have chosen to use a very basic FOM that allows us to gradually introduce more and more advances concepts. It is possible that part two of the tutorial may contain overviews of some commonly used FOMs.

6. Conclusions

A freely available HLA Tutorial has been produced. The tutorial is aimed at the practitioner. A software package with sample federates as well as free RTI and OMT software is also available.

Among the most important features of this new tutorial is that it presents HLA as a modern, service-oriented architecture. Another feature is to present the design chain that starts with the Federation Agreement, continues to the FOM and finally uses the HLA services. A third feature is to teach HLA based on tasks, like sending an interaction, rather than on the structure of the standard.

A large number of best practices have also been incorporated in the tutorial. These range from design and architectural practices down to low-level programming and testing.

We believe that the tutorial will have a positive impact on the adoption of HLA over the coming years, not only in the defense domain, but also in civilian applications.

References

[1] "The HLA Tutorial", www.pitch.se, September 2012

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

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

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

[5] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, August 2010.

[6] Frederick Kuhl, Richard Weatherly, Judith Dahmann: "Creating Computer Simulation Systems: an Introduction to the High-Level Architecture", Prentice Hall PTR (2000), ISBN 0130225118

[7] DMSO: "RTI 1-3-Next Generation Programmer's Guide Version 3.2", September 2000, US Departement of Defense: Defense Modeling and Simulation Office

[8] "Porting a C++ Federate from HLA 1.3 to HLA 1516" & "Porting a Java Federate from HLA 1.3 to HLA 1516", March 2003, http://www.pitch.se/support/pitch-prti-1516

[9] "Migrating a Federate from HLA 1.3 to HLA Evolved", November 2010, http://www.pitch.se/technology/about-hla-evolved

[10] "FEAT PDG - Federation Engineering Agreement Template", www.sisostds.org

[11] SISO: "Real-time Platform Reference Federation Object Model 2.0 ", SISO-STD-001 SISO, draft 17, www.sisostds.org

Author Biographies

BJÖRN MÖLLER is the Vice President and co-founder of Pitch Technologies. 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 M.Sc. 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 Support Group and the chairman of the SISO Real-time Platform Reference FOM PDG.

STEVE ERIKSSON is a software developer at Pitch Technologies and has been involved in the development of several commercial HLA products. He received his BSc degree in Computer science from Linköping University in Sweden.

ÅSA WIHLBORG is a Systems Developer at Pitch Technologies and a major contributor to commercial HLA products such as Pitch Visual OMT 2.0. She studied computer science and technology at Linköping University, Sweden.

Practical Experiences from Four HLA Evolved Federations

Björn Möller, Pitch Technologies, Sweden
Filip Klasson, , Pitch Technologies, Sweden
Björn Löfstrand, Pitch Technologies, Sweden
Per-Philip Sollin, Pitch Technologies, Sweden

bjorn.moller@pitch.se
fillip.klasson@pitch.se
bjorn.lofstrand@pitch.se
per-philip.sollin@pitch.se

Keywords:
HLA, HLA Evolved, Standards, Viking, SISO Smackdown, NATO MSG-068

ABSTRACT: *HLA Evolved was formally published in 2010 but early federations based on this standard have been developed since 2008. This paper summarizes experiences from four federations during the period 2009 - 2011 with focus on maturity and on the use of new HLA Evolved features. The federations are as follows:*

Viking 11 is the world's premier joint civil-military-police exercise involving more than 2500 persons from 31 nations distributed across nine sites. The purpose is to acquire hands-on practical skills and knowledge of civil-military-police coordination and cooperation before deployment in multifunctional and multinational UN mandated peace operation. The exercise was successfully carried out in April 2011 using an HLA Evolved infrastructure with participants running federates in different countries, proving the maturity of HLA Evolved.

NATO MSG-068 had the purpose to create a reference federation architecture for the NATO Education and Training Network with participants from, the US, UK, France, Germany, the Netherlands, Spain, Australia, Bulgaria, Romania, Turkey and Sweden. Participating systems include JCATS, JTLS, ICC, VBS2, WAGRAM, ORQUE, KORA and more. The HLA Evolved infrastructure proved to be stable during the experiment. HLA Evolved FOM Modules were used to mix standardized FOM data, like RPR with NATO extensions.

SISO Smackdown is a university outreach program, with participation from NASA and universities worldwide. Based on a lunar mission scenario it allows universities to extend the common information model to fit their part of the scenario. This extensibility is based on the HLA Evolved modular FOMs. Since this project uses RTIs from two different vendors it also illustrates how federations can benefit from the HLA Evolved Dynamic Link Compatible APIs.

BAE Systems Command and Control Demo Federation. This is a proof of concept for fault tolerance and load balancing. It uses fail-over federates based on the new HLA Evolved functionality. Another feature is the use of HLA Evolved Smart Update Rate Reduction for dynamic control of tactical data update rates on mobile devices.

The conclusion is that HLA Evolved is mature and that most of the new features have already been successfully used.

1. Introduction

This paper focuses on how the new version of HLA, formally called IEEE 1516-2010 and informally known as HLA Evolved [1], has been used in real life. It gives some background on why standards need to be maintained over time. It summarizes the new technical features of HLA Evolved. It then looks at how HLA Evolved and the new features have been used in some federations and provides some discussions on this

1.1 Evolving standards

Technical standards need to be maintained for several reasons.

- Solutions building on standards have evolving requirements. As an effect of this, the requirements on a standard and the features that it provides also needs to evolve.
- As technology in general develops, there are also new ways to implement technical features and new types of functionality that can be provided. Standards need to take advantage of this to improve or they will be replaced by newer standards.
- As different people implement a standard they may interpret it differently. This contradicts the purpose of standards, so this needs to be resolved.

- There will always be typos and inconsistencies in a standard that need to be corrected.

HLA was developed as a successor to DIS [2] and ALSP to provide one common architecture for simulation interoperability within the US Department of Defense. After a number of prototype federations the first complete version, HLA 1.3 [3] was released in 1996-1998 as a US DoD standard.

HLA was then taken to IEEE to become an open international standard. It was foreseen that there would be considerable synergies with sharing a standard for simulation interoperability with the civilian market. An international standard would also be advantageous when interoperating with coalitions partners outside of the US. The result was an HLA version called IEEE 1516-2000 [4].

All IEEE standards need to be opened for review regularly. They can then be revised, reconfirmed as is or retired. When the HLA standard was opened for review it had been used extensively in many projects. This feedback resulted in a large number or updates. The next (and current) HLA versions was developed by SISO and released by IEEE in August 2010 and it was formally named IEEE 1516-2010.

The version numbering of HLA (1.3, 1516-2000, 1516-2010) is somewhat confusing. If HLA were to be released as a commercial software product it can be argued that HLA 1516-2000 would have been called HLA 1.4 (minor upgrade) and HLA 1516-2010 would have been called HLA 2.0 (major upgrade).

1.2 Main technical improvements from HLA 1.3 to HLA 1516-2000

HLA 1516-2000 adds the following new features when compared to HLA 1.3:

- The Data Distribution Management (DDM) services where redesigned for greater flexibility. In HLA 1.3 dimensions were grouped into predefined routing spaces. Starting from HLA 1516-2000 a set of dimensions can be defined for an attribute or an interaction. These can then be freely combined at runtime as part of subscriptions or updates or interactions.
- The OMT format, used to specify FOMs and SOMs used, at that time, new XML Format, including a DTD definition.
- One update not to be underestimated was the introduction of fully specified data types in the OMT. One major cost driver during federation integration is unclear or misunderstood data representation when exchanging information.

Numerous other changes and clarifications were also made. One particular interesting one was a change in the semantics of publish and subscribe, making them additive instead of replacing, which in practice affected very few federates.

1.3 Main improvements from HLA 1516-2000 to 2010

A large number of updates were made to the HLA standard in the 1516-2010 version, both editorial and technical. Another paper [5] summarizes the technical updates in detail. This paper focuses on larger technical updates. These include:

- Modular FOMs [6] that enables a FOM to be built from modules. A module can contain objects, interactions and data types that relates to a well-defined sub-problem, like radio communications or federation management. This enables not only development and maintenance on a module level but also reuse and standardization in a modular and composable fashion.
- XML Schemas [7] for compliance testing of a FOM or a SOM replaced the previous DTD. A majority of the OMT specification requirements are covered by these XML Schemas. Some standards requirements cannot be checked by an XML Schema, for example multiple subclasses with the same name in the object or interaction class hierarchy.
- Fault Tolerance [8] support that includes a well-defined semantics for the RTI to signal to a federate if it has lost connection to the federation or if another federate has been lost.
- Web Services API [9] that enables a federate to call the RTI using Web Services instead of the C++ or Java API.
- Smart Update Rate Reduction [10] (SURR) that enables a federate to subscribe to updates with a particular upper threshold on the update rate.
- Improved Data Logging support that enables a federate to determine which federate that sent a certain update or interaction
- Standardized Time Representations [11] which provides two standard ways to represent time stamps independent of which RTI implementation that is used. One representation is an integer and the other is a float.

- Evolved Dynamic Link Compatibility (EDLC) for the C++ and Java APIs or the RTI. This enables a federate to use different RTIs without recompilation.

1.4 Sample federations

We will now look at a number of sample federations using HLA Evolved. Four major federations that use HLA Evolved and that are known to the authors have been selected. A few additional cases have been added to provide more examples. Today there are RTI implementations for HLA Evolved from at least three major RTI vendors (MÄK, Pitch and Raytheon-VTC) so nobody knows the total number of HLA Evolved RTI licenses in use and the number of HLA Evolved federations that are completed or under development.

2. Viking 11

2.1 Purpose

Viking [12] is the name of a series of combined civil-military crisis response operations exercises that have been carried out since 1999 under a joint Swedish and US initiative. The 2011 exercise focused on planning and conducting a United Nations mandated Chapter VIII peace operation/crisis response operation. The training event had about 2500 persons from 31 nations involved. The event involved not only national defense forces and blue-light organizations but also 35 non-governmental organizations (NGOs).

One of the more important requirements was to use a comprehensive approach, which is when military; police and civilian personnel operate together towards a common goal. The main focus was on CIMIC (Civilian-Military Cooperation) and command and control training. This includes using real operational command and control systems wherever possible. Computer based simulation systems were used to simulate the environment and scenario in order to stimulate the training audience.

The exercise builds upon a scenario called Bogaland [13], which has been developed in Sweden over a long time specifically for this type of exercise. It involves several fictional areas (East & West Kasuria, East & West Mida and Gotland) with several ethnic groups, uneven distribution of wealth, areas rich in natural resources like oil and gas and other areas with illegal economic systems. There are different ethnic and religious groups and the different governments are not in full control of their respective countries. There are a wide variety of issues to deal with, including protection of civilians, fighting piracy and hostile elements, dealing with irregular forces, reconstruction work and new developments as well as handling children taking part in armed conflicts.

The exercise ran for two weeks in April 2011 with a large number of training systems interoperating using an HLA Evolved infrastructure. The federation was stable for the duration of the exercise although a few of the participating systems had temporary issues. The official conclusion from

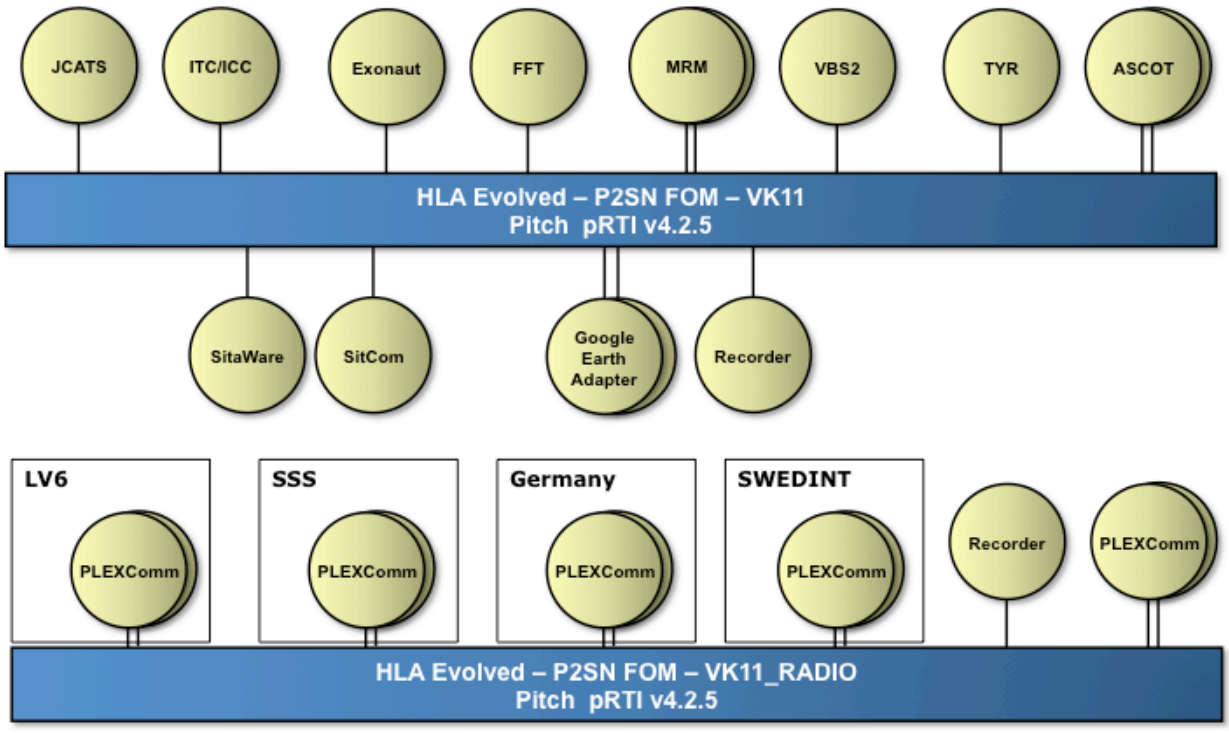


Figure 1: Viking 11 main federation and radio federation

the exercise was that *"It has been a great success and many innovations have been a first"*. This exercise was also mentioned in the ITEC 2011 keynote as *"the world's premier civil/military exercise"*.

2.2 Federation

The federation included systems both for the training audience and for the exercise control. The training audience was supported by systems providing a common operational picture, recognized air, land and maritime pictures, a tactical unmanned aerial vehicle view, civil/police situation awareness views, communication, collaborative tools, map tools and more. Systems for exercise control provided overall exercise management, *"God's Eye"* views, situational awareness, communication, collaboration tools and more.

A number of training systems were used including operational C2 systems, standard training systems and tailored systems. Some of the systems used include:

- JCATS, a well-known simulator, sponsored by US JFCOM and developed by LLNL, used for aggregated and platforms, regularly used for example by the US Marines.
- ICC, the operational NATO C3 system for showing and handling a recognized air picture for military air traffic controllers.
- ITC, a platform-level simulator owned by NATO, based on the Ternion Flames framework, that generates the air picture for ICC.
- Exonaut, a COTS, web-based, exercise management and planning tool used in the MEL/MIL approach, developed by 4C Strategies.
- FFT, the Friendly Force Tracker, a tailored application developed by Pitch, reports ground truth data from specific platforms or aggregates to C2 systems.
- MRM, the Multi Resolution Model, a tailored application developed by BAE Systems C-ITS, de-aggregates an aggregate unit into platforms.
- VBS2, a COTS game-based training system on the platform level by Bohemia Interactive, used to visualize the current situation from UAV and helicopter views.
- TYR, the major Swedish aggregate level war-gaming system from BAE Systems C-ITS.
- ASCOT, a training system for military air traffic controllers. This is a COTS application by PLEXSYS.
- SITAWARE HQ and WEBCOP, an operational C2 system is a COTS product from Systematic. It is regularly used by the armed forces of Sweden, Denmark, Finland, Germany and several other countries.
- SitCom, a tailored application developed by BAE Systems C-ITS, reports perceived truth from aggregated units in TYR to the federation.
- Google Earth PRO, connected to the federation through the COTS product Pitch GE Adapter. It was used for reporting situation awareness for blue-light and civilian participants that in many cases do not have their own C2 systems.
- PLEXComm, a radio communication simulator used to handle radio traffic exchange during the exercise. This is a COTS application by PLEXSYS.
- A FOM-agnostic data logger, Pitch Recorder, was used for collecting all Federation data exchange from the different Federations of the exercise.
- A COTS product for bridging and data filtering federate, Pitch Extender, was used to limit data exchange and to connect multiple federations.

The training systems were distributed across nine sites and the scenario included between 100 000 and 1000 000 entities. Many of these entities were dynamically aggregated and de-aggregated when needed, resulting in only 10 000 to 15 000 entities being registered in the federation at any single moment.

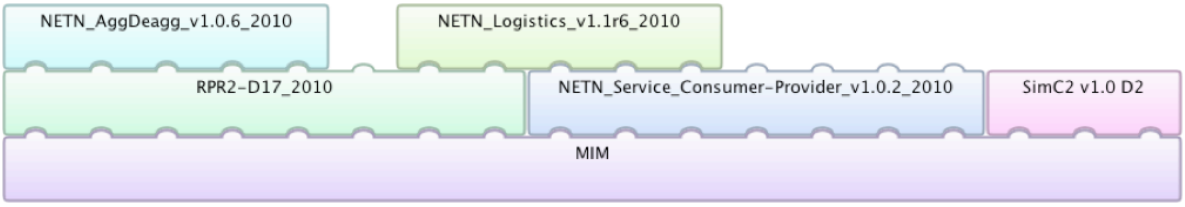


Figure 2: Viking 11 FOM modules

2.3 HLA Evolved aspects and experiences

The most important observation made here is that it is possible to carry out possibly the world largest comprehensive exercise with thousands of participants in different locations using HLA Evolved. The scenario was large, many operational systems from several providers were used, and the exercise ran for two weeks. This proves that HLA Evolved and many implementations (RTIs, tools, federates) have reached maturity for widespread deployment. The RTI used, Pitch pRTI Evolved was further optimized and tailored for large federations and severely memory-constrained federates during the early integration tests.

The federation used HLA Evolved Modular FOMs. The FOM Modules used were mainly the same as in the MSG-068 project described below, plus an additional module, SimC2 ("simulator to C2"). This set of FOM Modules is also known as the P2SN FOM version 2 where P2SN stands for Persistent Partnership Simulation Network, a cooperation between US JCW and the Swedish Armed Forces.

The federation had a mix of federates using the older HLA 1.3 and HLA 1516 APIs, as well as some using the new HLA Evolved APIs. While this is sub-optimal in the long run it makes it possible to quickly start reusing older systems in an HLA Evolved context.

Some of the tailored federates, namely the FFT, the MRM and the SitCom, were developed using a middleware generator, Pitch Developer Studio [14], which given a FOM generates middleware that supports both HLA Evolved and older HLA versions without recompilation.

3. NATO MSG-068

3.1 Purpose

The mission of the NATO Modeling and Simulation Group (MSG) is to promote co-operation among Alliance bodies, NATO member nations and partner nations to maximize the effective utilization of M&S. NMSG is part of the NATO Research and Technology Office (RTO). The group NMSG-068 focused on developing recommendations for a NATO Education and Training Network, NETN. Such a network should *"integrate and enhance existing national capabilities and focus on the education and training of NATO Headquarters' staffs and NATO forces. A NETN consisting of a persistent infrastructure, distributed training and education tools, and standard operating procedures not only supports the training of NATO headquarters but also enables the Nations to collaborate with each other"*.

MSG-068 [15] had more than 140 experts collaborating, representing Joint Warfare Center, Joint Force Training Center, NC3A, ACT, US JFCOM, Australia, Bulgaria, France, Germany, Hungary, The Netherlands, Romania, Spain, Slovenia, Sweden, Turkey and the UK.

The project developed a federation agreement and a set of FOM modules for a NETN. A number of design patterns were developed, for example how entities in one simulator can requests services, for example logistics, from another simulator. Another pattern was the modeling of transfer of control for resources and entities between different simulations.

More than eight major integration and test events took place with different combinations of systems and focusing on different tasks such as convoy, repair, supply patterns and more. This lead up to a main experiment in October 2010 involving multiple training centers and national battle-labs and a demonstration at I/ITSEC 2010.

The work of MSG-068 continues in MSG-106 with additional focus on operational requirements and support to CAX. A lot of new developments in M&S in NATO build upon the MSG-068 work;

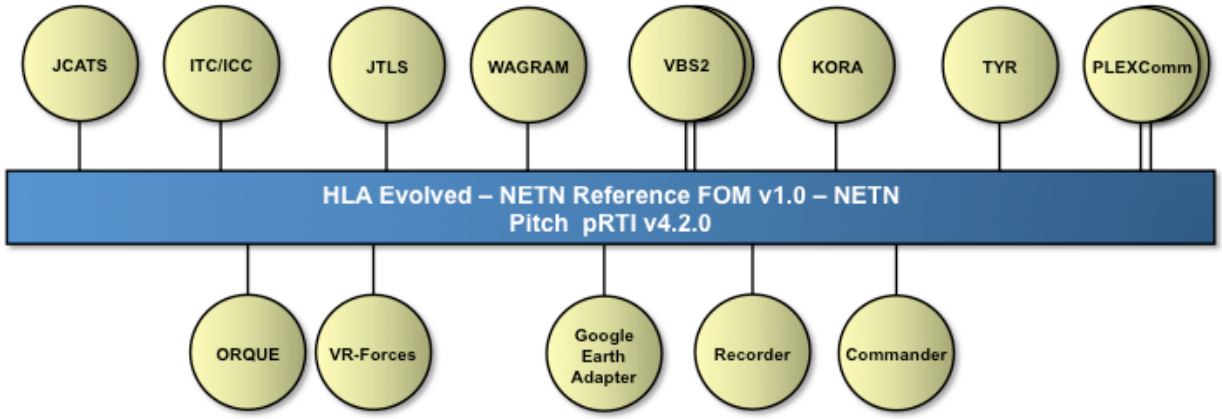


Figure 3: NATO MSG-068 federation

Viking 11 (above) is one example of this.

3.2 Federation

A number of experimental federations were developed and executed as part of the project. Examples of participating NATO and national systems and tools are: JTLS, JCATS, JPECT (ITC-FLAMES/ICC), VBS2 through LVC Game, FACSIM, MäK VR-Forces, MäK Stealth Viewer, WAGRAM, ORQUE, ALLIGATOR, PLEXComm, KORA, PsiWeb, Marcus, CATS TYR, Netscene, Google Earth PRO, Pitch GE Adapter, Pitch Commander, Pitch pRTI Evolved, Pitch Booster and Pitch Recorder.

3.3 HLA Evolved aspects and experiences

This project started using early versions of HLA Evolved. It was one of the first to work with modular FOMs in practice. There were minor fixes to the HLA Evolved APIs during this project but no major changes in the functionality. By the end of this project there were not only a stable RTI for HLA Evolved available but also a number of general purpose HLA Evolved tools.

This project pioneered the work with full-scale FOM module development. The set of FOM modules developed consists of a base with the RPR-2 FOM draft 17 [16]. It adds a service consumer/provider module and on top of this a Logistics module. A Link-16 [17] module (described as a "BOM") is included. A special Federation Execution Management module was also developed. A special module for aggregation and de-aggregation was also developed.

The federation also had a mix of federates using the older HLA 1.3 and HLA 1516 APIs as well as the new HLA Evolved APIs.

MSG-068 also contribute to the continued work with standardizing RPR-FOM v2.0 by providing a proposed set of HLA FOM modules that represents the full RPR-FOM v2.0 D17 object model to the SISO RPR-FOM PDG.

4. SISO Simulation Smackdown

4.1 Purpose

SISO Simulation Smackdown [18] is an outreach program where university students are invited to participate in building HLA federations together with NASA, one of the sponsors of the project. The purpose is to promote modeling and simulation subjects in education. NASA provides some basic federates and HLA vendors (ForwardSim, MÄK and Pitch) provide HLA tools and RTIs. Aegis provides coordination and mentoring. SISO host locations and provides coordination and funding. Participating universities at the 2011 Smackdown came from North America, Europe and Asia.

4.2 Federation

The scenario is space mission support in the proximity of the Earths moon. Participating federates include:

- An environmental federate with the position for the Sun, Earth and Moon as well as a federate with a simple transfer vehicle, provided by NASA

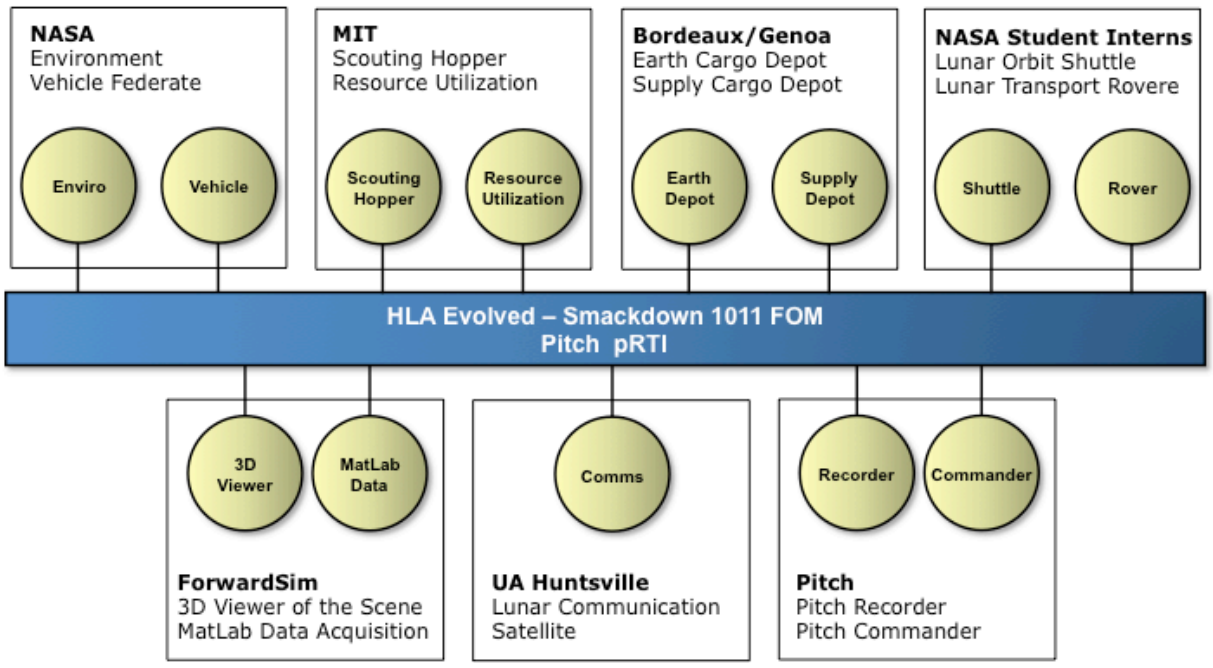


Figure 4: SISO Smackdown federation

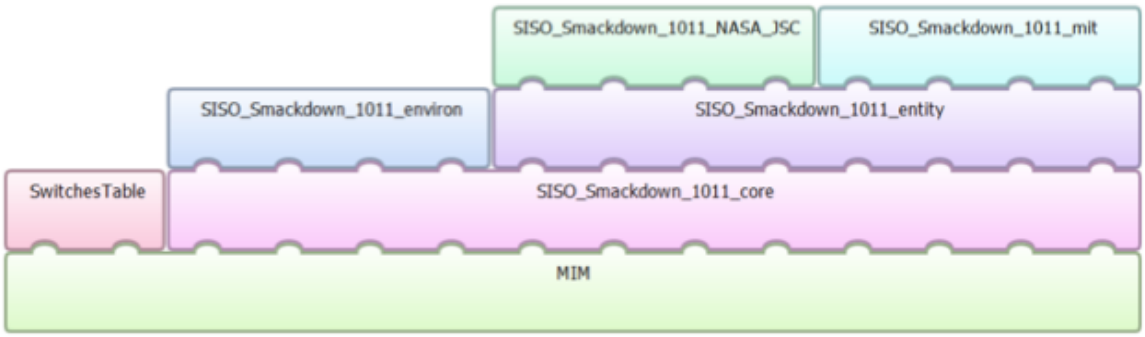


Figure 5: SISO Smackdown FOM modules

- Earth cargo depot and supply cargo depot federates from the University of Bordeaux/Genoa team.
- Lunar communication satellite federate from the University of Alabama Huntsville team.
- A high-mobility scouting hopper and a mobile in-situ resource utilization plant federate from the MIT team
- A lunar orbit shuttle and a lunar transport rover federate from the NASA student intern team
- A 3D viewer of the scene from ForwardSim
- Federates from Keio University of Japan, which unfortunately had to withdraw due to the events in Japan.
- Data loggers from MÄK and Pitch

This federation used HLA time management.

4.3 HLA Evolved aspects and experiences

This federation used several HLA Evolved features. First of all the modular FOM approach was used with three common FOMs: the Core FOM, the Environment FOM and the Entity FOM. Students teams that needed additional concepts added more FOM modules.

Since this federation used both the MÄK and the Pitch RTI the Evolved Dynamic Link Compatible (EDLC) APIs were tested in practice. Some federation executions were done using the MÄK RTI and some using the Pitch RTI.

The federation also used an HLA Evolved standardized time representation.

5. BAE Systems C2 Demo

5.1 Purpose

A group within BAE Systems has developed a fault-tolerant, low-bandwidth training concept. It has been demonstrated for a number of customers. It contains several solutions for common problems in C2 training exercises on the tactical/platform level. It uses a number of interesting features based on HLA Evolved. One of them is fault tolerance. Another one is the ability to do load balancing between simulations.

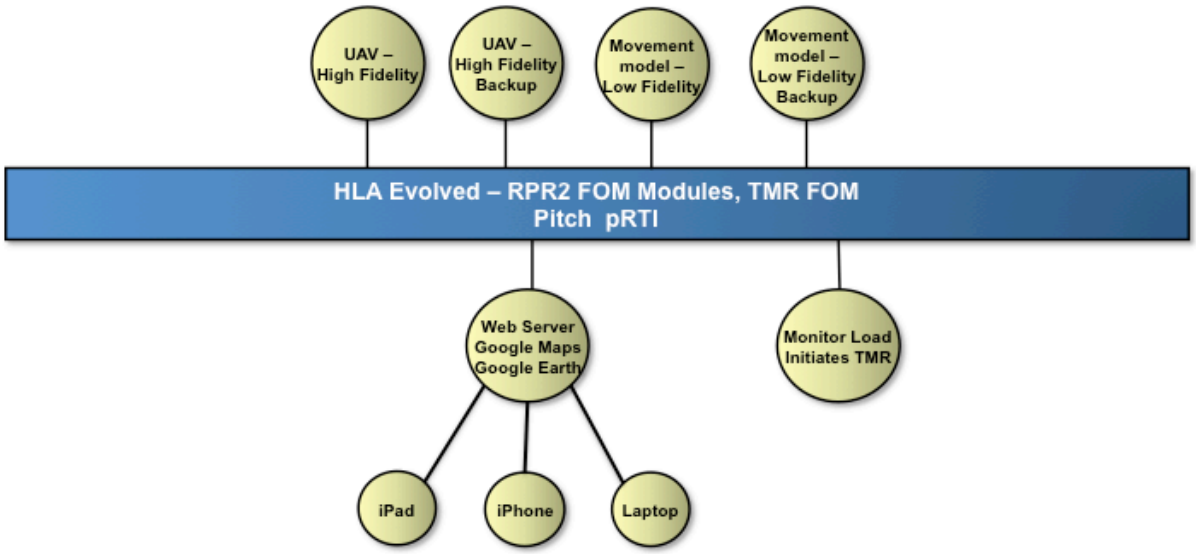


Figure 6: BAE Systems C2 Demo – before fail-over

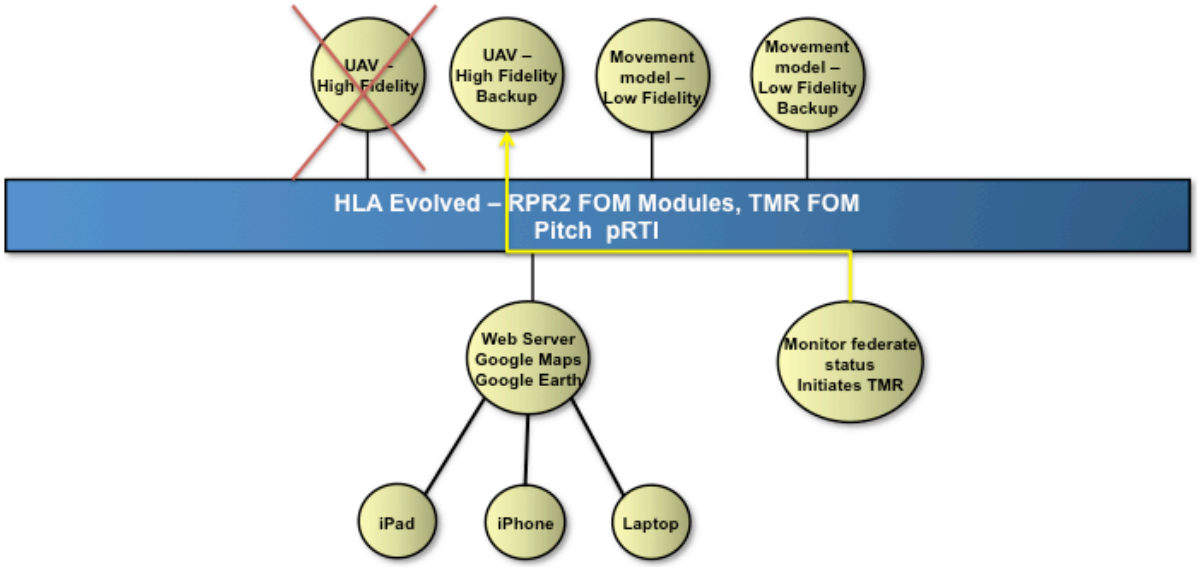


Figure 6: BAE Systems C2 Demo – after fail-over

5.2 Federation

The federation uses the RPR2 FOM for representing platforms as well as a pattern for Transfer of Modeling Responsibility (TMR) that will be discussed further in NATO MSG 106. Each federate provides data to the federation about its capabilities for simulating different types of platforms as well as the fidelity level of the simulation. The CPU and memory load of the federates is also provided to the federation. This makes it possible to handle fail-over situations where it is necessary to find a new simulator that can model the behavior of platforms from federates that have been lost. All federates in the federation are rated based on their simulation capability, simulation fidelity and their CPU and memory load and the federate with the best score is selected as the new owner. The process of finding a new owner is automatic by default but it is possible for an operator to manually select a new owner of an entity from a Google Maps Web interface.

When the lost federate joins the federation again the monitoring federate initiates the Transfer of Modeling Responsibility to return ownership to the original owner.

The federation consists of the following federates:

- A mapping and controller federate. It shows platform positions. It also enables the operator to request that another federate takes over the responsibility for a platform.
- A number of simulators that can model platforms according to the RPR2 FOM, for example UAV simulators.
- A load balancing and fault tolerance management federate.

5.3 HLA Evolved aspects and experiences

This federation uses modular FOMs. It also uses the fault tolerance semantics of HLA Evolved. Lost federates are detected using both the HLA Evolved "Federate Lost" interaction and by monitoring the disappearance of object instances.

The federation implements the following fault tolerance design patterns, described in a previous paper (05S-SIW-048).

- The reoccurring federate, meaning that the federate that has lost connection to a federation will periodically try to reconnect.
- The fail-over federate, meaning that there is a backup federate standing by to take over ownership of an instance.
- The fault-monitoring federate, that subscribes to MOM information and monitors and assesses the state of the federation, in particular with respect to participating federates.

This federation also uses the HLA Evolved Smart Update Rate Reduction (SURR) to reduce the update rate for platform position updates to federates with limited bandwidth or processing power.

6. Some Other Noteworthy HLA Evolved Work

6.1 German NAVY FOM

The new German Maritime Federation Object Model [19] aims to prove interoperable simulation capabilities for the German naval command and control systems. Important federations to interoperate with include the US Navy NCTE, the

US Joint JLVC, NATO federations and the German Forces distributed network SuTBw. The work has been initiated, guided and performed by several organizations including the German Modeling and Simulation Commissary, The Combat Direction Systems Activity (CDSA), the NavCCSysCom and the Novonics Corporation.

The object model uses HLA Evolved FOM modules. It is based on the RPR FOM [16] version 2d17 and the project considers aligning with future, standardized version of the RPR FOM. Additional FOM modules cover areas like Link 16 communications, emitters and transponders, sonobuoys, acoustics and more.

6.2 Object Oriented Middleware Generator using the Web Services API

Pitch Developer Studio [14] generates HLA middleware in C++ or Java for a particular FOM or a set of FOM modules. This middleware can then be integrated into a simulation that requires HLA capabilities. The middleware then provides C++ or Java objects for all instances in the federation, including type-safe "setters" and "getters" for attributes and parameters.

The generated code is compatible with the HLA Evolved, 1516-2000 and HLA 1.3 APIs from most leading RTIs using either the C++ or Java APIs, without recompilation. A new feature is that a federate developed using Pitch Developer Studio also supports the Web Services API of an RTI. This means that a federate can participate using the C++ API one day and in another federation using the Web Services API the next day.

6.3 Data logging in Pitch Recorder

Pitch Recorder [20] is a data logger that supports HLA Evolved, HLA 1516-2000 and HLA 1.3, as well as DIS, Voice and user-defined formats. All formats can be recorded in parallel and in sync. When recording data from an HLA Evolved federation it can use the new HLA Evolved data logging features. It is possible to specify that a particular Pitch Recorder channel shall only record data (object registrations, attribute update, interactions) from a particular named federate. FOM modules are also supported.

6.4 FOM Development in Pitch Visual OMT

Pitch Visual OMT [21] is a FOM and SOM development tool that has been used in most of the above projects. It has several ways to check the correctness of an HLA Evolved FOM or a set of FOM modules. It can use the HLA Evolved Schemas to check a FOM for example to verify that it contains all necessary data to initialize a federation execution. In addition to this it provides a built-in rule engine that checks a set of FOM modules for problems on three levels:

- Error, meaning that there is a problem in this set of FOM modules. They do not contain enough information to initialize a federation execution.
- Warning, meaning that there is a minor problem that needs to be corrected, for example a missing definition of a data type.
- Information, meaning that it doesn't follow best practice, for example by not documenting the semantics of an attribute or not providing a point of contact.

7. Feature Usage Summary and Discussion

This section attempts to summarize the usage of HLA Evolved and the new features in particular.

First of all it is important to notice that almost all of the new HLA Evolved features have been successfully used in practice, even though the standard was published in August 2010, which is a year and a half ago. It is also worth noting that HLA Evolved has already been used in such a large and demanding federation as Viking 11, which indicates the maturity of both the standard and some RTI implementations.

Modular FOMs have been use in more or less all federations. They improve the development and maintenance of almost any federation, large or small. The need for modification of existing federates is very limited.

Fault Tolerance support is a highly requested feature for many federations that have grown in size or that needs to be deployed under less than perfect conditions. This requires a redesign of the federation agreement and in some cases major modifications of the federates that may need to run in both connected and disconnected mode and to support late joiners. Nevertheless one federation has successfully used it.

The Evolved Dynamic Link Compatibility (EDLC) features of HLA Evolved have been tested in the Smackdown federation. This federation is quite small but it uses a wide range of HLA Services, including Time Management. This makes it a good test. EDLC is important to build a good marketplace where federation can easily switch between RTIs and GOTS or COTS federate can be expected to work with any compliant RTI.

The Smart Update Rate Reduction (SURR) has been used in one federation but could have been useful in some of the other federations to reduce the load on selected federates.

Standardized time types are important for the reuse of time managed federates. This list contains only one time-managed federation, Smackdown, which does indeed use such a representation. Note that the tools Pitch Recorder and Pitch Developer studio also supports this. It is very hard to develop COTS tools that support non-standard time representation since a tool needs not only to receive the unknown data but also to interpret and process it.

Improved support for data logging has only been used in one of the examples, namely the data logger. Still any federation can use this HLA Evolved feature without modification since it only requires the acquisition of a data logger with such support.

8. Conclusions

A number of federations have been already been developed using HLA Evolved. The have used almost all of the major new features. All new HLA Evolved features have worked as expected.

Large exercises with thousands of participants have been successfully performed using HLA Evolved infrastructures, which proves the maturity of several implementations.

The most important conclusion is that the new features of the HLA Evolved standard match a number of actual requirements of simulation and federation users.

References

[1] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, August 2010.

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

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

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

[5] 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.

[6] Björn Möller, Björn Löfstrand. "Getting started with FOM Modules", Proceedings of 2009 Fall Simulation Interoperability Workshop, 09F-SIW-082, Simulation Interoperability Standards Organization, September 2009.

[7] W3C. "XML Schema Definition Languaged (XSD)", 5 April 2012, http://www.w3.org/TR/xmlschema11-1/, accessed 10 April 2012

[8] Björn Möller, Björn Löfstrand, Mikael Karlsson. "Developing Fault Tolerant Federations using HLA Evolved" Proceedings of 2005 Spring Simulation Interoperability Workshop, 05S-SIW-048, Simulation Interoperability Standards Organization, April 2005.

[9] 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.

[10] Björn Möller, Mikael Karlsson. "Developing well-balanced federations using the HLA Evolved smart update rate reduction". Proceedings of 2005 Fall Simulation Interoperability Workshop, 05F-SIW-87, Simulation Interoperability Standards Organization, September 2005.

[11] Mikael Karlsson, Fredrik Antelius, Björn Möller, "Time Representation and Interpretation in Simulation Interoperability – an Overview", Proceedings of 2011 Spring Simulation Interoperability Workshop, 11S-SIW-049, Simulation Interoperability Standards Organization, April 2011.

[12] Viking 11, http://www.forsvarsmakten.se/en/About-the-Armed-Forces/Exercises/Completed-exercises-and-events/VIKING-11/

[13] Swedish Armed Forces International Centre, UN-Courses, "Background of Bogaland", http://www.forsvarsmakten.se/upload/Forband/Centra/Forsvarets_internationella_centrum_Swedint/RAP_A1_01_Background_of_Bogaland%20_111_BAF.pdf, accessed 1-Mar-2012.

[14] Björn Möller, Fredrik Antelius. "Object-Oriented HLA - Does One Size Fit All?", Proceedings of 2010 Spring Simulation Interoperability Workshop, 10S-SIW-058, Simulation Interoperability Standards Organization, April 2010.

[15] Björn Löfstrand, Rachid Khayari, Konradin Keller, Klaus Greiwe, Peter Meyer zu Drewer, Torbjörn Hultén, Andy Bowers, Jean-Pierre Faye. ”Logistics FOM Module in Snow Leopard: Recommendations by MSG-068 NATO Education and Training Network Task Group”, Proceedings of 2009 Fall Simulation Interoperability Workshop, 09F-SIW-076, Simulation Interoperability Standards Organization,September 2009.

[16] SISO, “Real-time Platform Reference Federation Object Model 2.0 ”, SISO-STD-001 SISO, draft 17.

[17] Link 16 is defined as one of the digital services of the JTIDS / MIDS in NATO's Standardization Agreement STANAG 5516. MIL-STD-6016 is the related United States Department of Defense Link 16 MIL-STD.

[18] SISO Smackdown web page, http://sisosmackdown.com, accessed 1-Mar-2012.

[19] Kay Roos, Ilja Olomski, Peyton Campbell, Brian D. Mack, Steven L. Smith, Richard Rheinsmith, “A Multi-Faceted Approach to the Development of the HLA 1516-2010 German Maritime Federation Object Model (GMF)”, Proceedings of 2012 Spring Simulation Interoperability Workshop, 12S-SIW-048, Simulation Interoperability Standards Organization, March 2012

[20] Björn Möller, Fredrik Antelius, Tom van den Berg, Roger Jansen, ”Scalable and Embeddable Data Logging for Live, Virtual and Constructive Simulation: HLA, Link 16, DIS and more”, Proceedings of 2011 Fall Simulation Interoperability Workshop, 11F-SIW-055, Simulation Interoperability Standards Organization, September 2011.

[21] Pitch Visual OMT web page, http://www.pitch.se/products/visualomt, accessed 1-Mar-2012.

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 M.Sc. 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 Support Group.

FILIP KLASSON is a Systems Developer at Pitch and is a major contributor to several commercial HLA products from Pitch. He holds an MSc in Computer Science and Engineering at Linköping University, Sweden.

BJÖRN LÖFSTRAND is the Manager of Modeling and Simulation Services at Pitch Technologies. He holds an M.Sc. in Computer Science from Linköping Institute of Technology and has been working with HLA federation development and tool support since 1996. Recent work includes developing federation architecture and design patterns for HLA based distributed simulation. He leads the MSG-068 FOM and Federation Design (FAFD) technical subgroup.

PER-PHILIP SOLLIN is a Software and Systems Developer and the Enterprise Integration Specialist at Pitch. He has been involved in the Development of several Pitch tools including the DIS Adapter, GE Adapter, Visual OMT, Commander, Booster and pRTI Evolved. He has also been working with Networks and Integration in many major exercises and experiments including Viking08, Viking11, CJSE12 and MSG-068. He studied Master of Science and Engineering in Software Engineering and Technology at Chalmers University in Gothenburg, Sweden.

Security in Simulation – A Step in the Right Direction

Stella Croom-Johnson – Dstl, UK
Wim Huiskamp – TNO, The Netherlands
Björn Möller - Pitch Technologies, Sweden

scjohnson1@dstl.gov.uk
wim.huiskamp@tno.nl
bjorn.moller@pitch.se

Keywords:
Security in simulation, simulation, training, NATO, security, requirements, DSEEP, information exchange, multi-level security, cross-domain solutions, accreditation

ABSTRACT: *A number of approaches are currently in use to allow limited sharing of data between simulations running at different native classification levels, but each have their associated issues which prevent full interoperability. This presents users and accreditors alike with a unique set of challenges. Building on the work presented to recent SIWs by NATO MSG 080 (Security in Collective Mission Simulation) the Security in Simulation Standing Study Group has been considering the role standards might play in making progress towards a Cross Domain Solution.*
This presentation summarises the work of the SSG showing how it has built on past papers and the work of NATO MSG-080 to identify where standards might contribute to – if not a full Cross Domain Solution – at least to making progress in this area.
The SSG members propose to draw on national use cases to create a set of guidelines for best practice, to create a taxonomy of terms commonly in use and to create a Security overlay for DSEEP. The paper will examine some of the use cases to consider how they might be applied across the various approaches, where they highlight common challenges and what this might mean for the proposed product nomination.
SISO cannot expect to influence the policies and processes of individual nations, but engagement with their accreditors is an important factor and it is hoped this paper will provide sufficient material to stimulate engagement and obtain their buy-in.

1. Introduction

In Joint Collective Training there is an increasing need to achieve simultaneous, multi-way interoperability between simulations operating at different native classification levels.
Interoperability standards (DIS, HLA, TENA, etc.) are already in place to connect the simulations based on 'ground truth' exchange of all relevant data, but to create an accurate representation of operational issues there is also a need to share certain information in accordance with the classification levels that are in place to protect that data.
A significant and growing percentage of training in the foreseeable future will be with coalition partners: this means that participating simulations need to be connected across not only domain boundaries, but also across national boundaries. To enable this, internationally agreed standards are needed to support a flexible and adaptable security architecture, which ensures that the appropriate interactions take place between the participating simulations without violating security classifications.
The problem space is now reasonably well understood, and the next step is to consider how to take things further using a combination of existing and novel processes and technologies.

2. Background

This is not a new challenge and as long ago as 1997 a paper to the SISO Fall SIW [1] mentioned the issues arising from the need to exchange data between systems operating at different security levels. In more recent years the topic was revisited in a presentation to the 2009 Spring SIW [2] which looked at the limitations arising from the conflict between the need to share data and the need to protect that same data, and outlined the concept of a labelling and release mechanism that could be applied to prevent leakage of sensitive information.

.

This paper takes a brief look at three subsequent papers [3] [4] [5] presented by members of NATO Modelling and Simulation Group MSG-080 to SISO in recent SIWs and shows how the SISO Security in Simulation Standing Study Group (SiS SG) has built on these to determine the role standards could play in making progress towards a Cross Domain Solution.

The MSG-080 papers (from which Figures 1-5 are reproduced) looked at a number of scenarios and use cases with typical solutions. Five possible approaches were outlined, four of which are in current use: the other is a vision of how a true Cross Domain solution might operate. These are covered in some detail in the papers so the summary given below is intentionally very brief.

2.1 System High

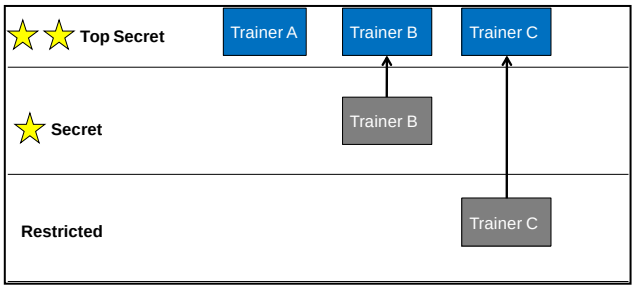


Figure 1: All systems and participants reclassified to highest level

Everything – including data and (potentially) facilities is reclassified to the highest level. This effectively means that each participant agrees to expose all data that is exchanged with all other participants. This may result in unacceptable risk for some participants leading to withdrawal from the exercise or significant rework to 'dumb-down' a classified simulation with possible loss of training value.

2.2 Multiple Single Levels of Security

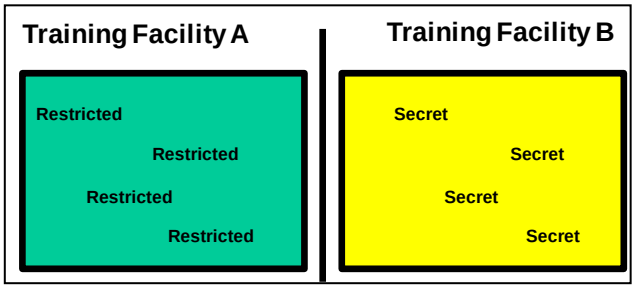


Figure 2: Physically separate domains

The security domains are physically separate, although limited data exchange can be achieved via manual intervention. Interactive response time will be limited and the burden of guarding against information leaks will fall on a human operator.

2.3 Multiple Independent Levels of Security

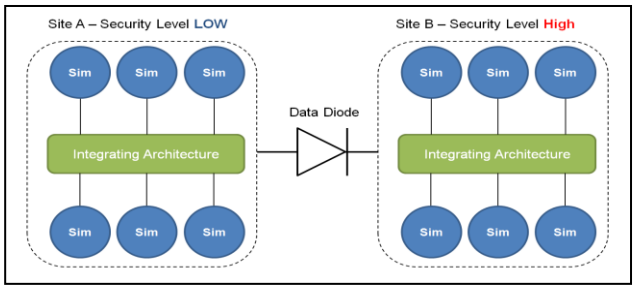


Figure 3: One way flow using a data diode

A data diode permits a unidirectional data flow from (in this case) Low to High, but there are no true two-way interactions between the domains. This may severely limit the training value. The approach is also rather blunt: there is no inspection or decision at the information level. All data is either passed or blocked based on source and destination.

2.4 Information Exchange Gateway

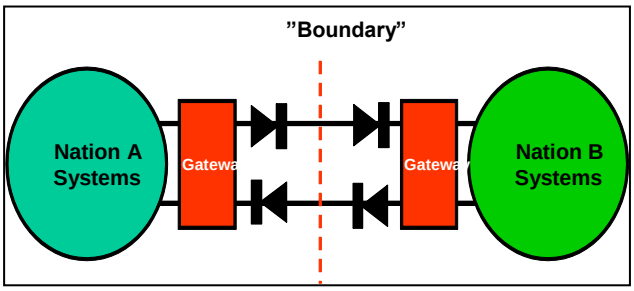


Figure 4: Limited 2-way data exchange across security boundaries

This is useful when multiple security authorities are involved who do not necessarily trust each other. Data is sanitized by using a combination of devices such as Data Guards and Data Diodes. This achieves a limited form of two-way interaction between simulations, but has the disadvantage that data discrepancies arise from the use of accurate data in some federates and sanitized data in others.

2.5 Trusted System (vision)

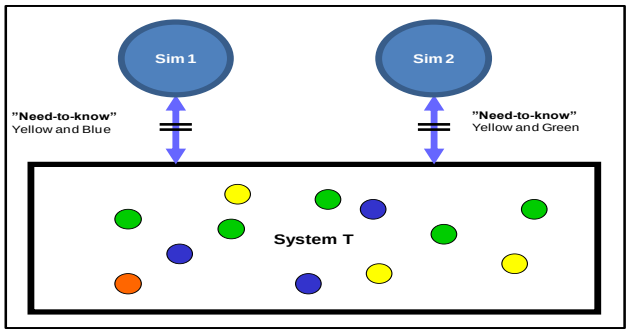


Figure 5: Data released on 'need-to-know' basis

In this approach data from systems of different classifications would be permitted to mix freely, using devices such as Data Diodes and Data Guards to release data to participants on a 'need-to-know' basis.

2.6 Context

A number of use cases around typical scenarios and solutions and early experiments were identified and analysed to show where issues might arise on the exchange of data within a simulation, where the sensitivities lie, and how this highlighted the main concerns, issues and threats.

Each of the approaches has their own strengths and weaknesses, but no formal guidelines exist to indicate under which circumstances one approach might work better than another.

The MSG-080 papers also identify where security for Collective Mission Simulations differs from that for other domains and concluded that the main areas are:

- Federates exchange accurate data (ground truth), which means it can – potentially – be accessed by all participants. In the real world the data released to a given individual depends on what can be directly observed, or is specifically released to that individual.
- In simulation a given scenario can be replayed, giving rise to a larger sample size than would be available in real life, with the opportunity to examine the simulation 'ground truth' data in slower time.
- Simulations require truly interactive, low latency levels of data exchange.
- The need to meet the goals of the simulation (e.g. effective training) may cause conflict with the security goals.

The papers concluded that a holistic approach is necessary, with a focus on risk management rather than risk avoidance or risk acceptance and with an overarching need to obtain early engagement from the accreditation communities involved in any given event.

A number of isolated strands of work are ongoing in various organizations/nations and the importance of ensuring coherency both between proposed solutions as the problem space evolves and with existing simulation standards forms part of the objectives of the Security in Simulation SSG.

3. SISO and Security in Simulation

The approaches outlined earlier are already well understood by both the simulation and accreditation communities, but they do highlight a number of issues:

- Each use case is very different from the next one: what constitutes an acceptable solution will vary from case to case and there is no 'one size fits all' solution.
- Local Subject Matter Experts (SMEs) often understand what works well in their own context, but this may not be transferrable to a distributed federation involving disparate players. Although individual organisations might have something written down this is not always the case at national or international levels.
- Any implementation will depend on what an accreditor is willing to approve – and each nation's accreditors have different perspectives.

The current process is designed to ensure compliance with national accreditation requirements. Whilst this manages the security issues (e.g. avoidance of data leakage) the implementation of it can create simulation related issues (e.g. how to achieve meaningful data exchange between simulations). The SiS SSG concluded that whilst new/amended standards would be of limited utility in this context other SISO products do have the potential to yield significant benefit during the development of a simulation, and to facilitate engagement with the accreditation community. The recommendations were:

a) The creation of a security overlay to DSEEP to help users consider the implications of security in simulation at each of the 7 DSEEP stages. This would highlight what is important to a given simulation and show where the challenges are likely to arise.

b) The creation of a 'Best Practice' Guide to provide a baseline from which to work when setting up a simulation: what has worked in the past, and – perhaps as importantly – pitfalls to avoid.

c) To create an agreed, common glossary for Security in Simulation to ensure all participants

have a common understanding of the terms used. To ensure coherence with existing glossaries and ontologies these would be used as a starting point.

4. Two use cases: (JTEN and MTMD)

4.1 UK – mapping JTEN to DSEEP

4.1.1 Use case Description

As part of a task to gain a better understanding of the potential utility of using the US Joint Training and Experimentation Network (JTEN) to link a simulation based in the US to one based in the UK, a series of trials – known as the 'JTEN' trials – took place in 2008.

In order to provide a useful training environment the trials used the JTEN network to link JFCOM in the US to Westdown Camp in the UK allowing UK and US participants to communicate over a node on the JTEN network.

The aim of the first trial was to allow a 'live' Forward Air Controller (FAC) on the range at Westdown Camp to direct a pilot in the US flying a simulated aircraft in a Close Air Support (CAS) role and for the resulting ground truth effects of any munitions dropped by that aircraft to be made visible – via Synthetic Wrap[1] (SW) - to the FAC on the ground using an Augmented Reality (AR) monocular to view the effects.

A live exercise on Salisbury Plain (at Restricted) ran concurrently with JTEN Trial 1. The AWES[2] system generated a feed from the exercise on Salisbury Plain into a Synthetic Wrap, allowing the simulated entities on both sides of the Atlantic to interact with the live entities on Salisbury Plain.

JSAF was used as a simulation in both countries and data was exchanged over the JTEN network by the transmission of DIS PDUs. The US simulation and the simulated part of the UK event ran at Secret, but the live exercise, the AWES system and the FAC had a Protective Marking of Restricted. A data diode permitted data to pass from Restricted to Secret, but no data could be passed the other way.

As a result of this, the AWES system did not receive detonation PDUs direct from the DIS network and the FAC was unable to receive information automatically about weapon effects originating from the Secret US simulation. To handle this, a terminal in the Restricted domain (showing the AWES 'ground truth' & which enabled the operator to manually inject detonation events into the AWES simulation) was located near to an equivalent terminal in the Secret domain. The 'air-gap' between the two systems was managed by an air gap operator (a 'man in a swivel chair'), who monitored the events in the Secret domain (i.e. location of detonation events) and manually replicated the ground effects through the manual triggering of detonation events within the AWES system, triggering the appropriate interactions, validating the detonation point location through the Secret system, as both (the original detonation point & the manual inject) were visible on the Secret system. Through the interface to AWES the information was sent to an AR monocular to allow the FAC to visualise the detonation event. The FAC was then able to communicate the results of the strike to
the pilot in the US simulation.

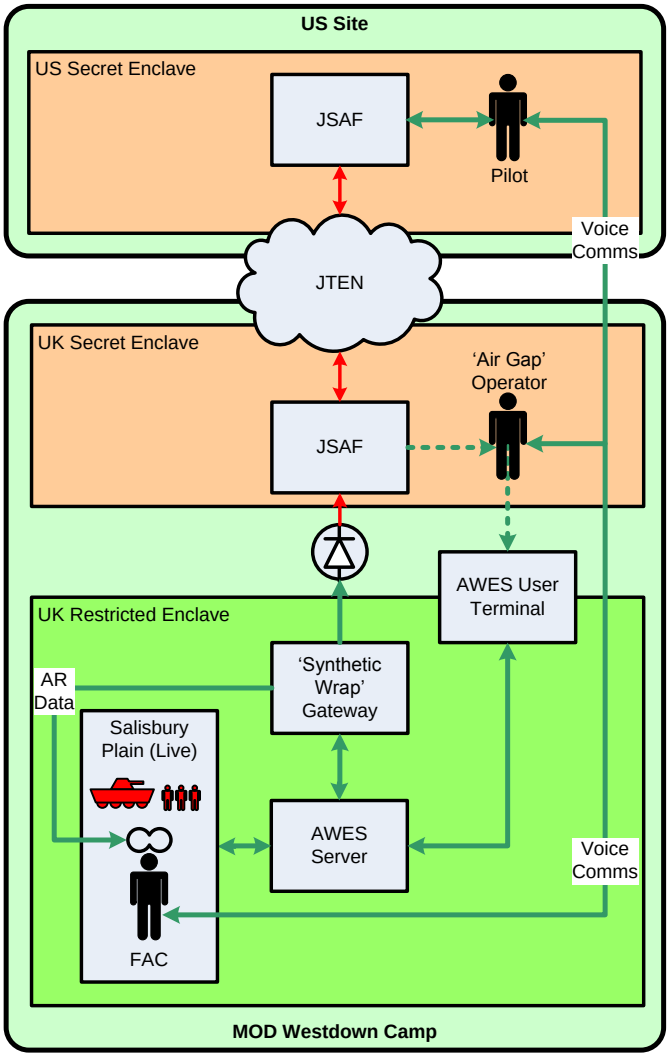


Figure 6: Architecture for JTEN Trial 1

[1] a data bridge between the 'virtual' simulation network & the 'live' tactical engagement simulation (TES)
[2] Area Weapons Effects System, provided by Cubic.

The flow of events was as follows:

- UK FAC guides US pilot to target.
- The US JSAF pilot released a munition.
- UK JSAF in the Secret domain received a DIS PDU from the US JSAF providing information about the detonation, displayed to the air gap' operator on the UK JSAF graphical map display. The Live entities on Salisbury Plain, and the FAC were outside the Secret enclave, so did not have visibility of this.
- A Restricted terminal, located adjacent to the Secret enclave, displayed events from within the Restricted domain as they occurred.
- The air gap operator manually replicated the detonation events into AWES: this generated feeds into the SW and the AR monocular.
    - The SW allowed the AWES system to calculate the outcome of the detonation event on the live players – i.e. whether they were 'killed' or 'damaged' or not affected.
    - The AR monocular allowed the FAC to visualise the outcome of the detonation, which meant he could inform the US pilot of the result using his radio.

As part of the After Action Review it was agreed that whilst this was a workable solution it was far from ideal. Inevitably, inaccuracies were introduced, delays were experienced and there were noticeable discrepancies between the simulations running at Secret, and the entities relying on manual injects.

One example of potential issues experienced relate to the targeting of a live vehicle. The delta between events in a simulation and manually replicated events are likely to mean that by the time a vehicle became aware of a detonation it could have travelled some distance from the point of impact. As it would no longer be at the point of detonation it would not realise it had been destroyed, so would continue to execute its mission. On the other hand, the co-ordinates from the simulation would show that the vehicle was hit – and destroyed – by the munition. This situation was avoided in the JTEN trials by keeping the enemy target vehicle static; obviously an artificiality for the trials which would not be acceptable for real training.

An alternative approach might have been to permit the vehicle to move and for the air gap operator to be responsible for keeping the domains in step. He would have been aware that the vehicle was shown as disabled in the Secret enclave, but not in the Restricted enclave. Manual intervention in the Restricted enclave would have then brought the two representations back into line with each other.

As part of the post-exercise discussions with the accreditors it was agreed that this situation could be improved. The DIS munitions PDU contains descriptors of the location and magnitude of a detonation, but no weapon or performance parameters are passed – neither data on the type of munition nor when it was released. On these grounds, the event accreditors gave a verbal indication that for future exercises of this type the detonation PDU might be transmitted into the Restricted enclave, but this has not yet taken place. It is also possible that a similar arrangement might be allowable for other PDU types provided they do not contain sensitive information but this would be subject to further discussion with the accreditors.

4.1.2 Use case mapping on DSEEP

The following paragraphs demonstrate how this use case might be mapped against DSEEP – the DSEEP steps are given in normal font, the mapping in *italics*:

Step 1: Define Simulation Environment Objectives

- Identify user/sponsor needs
  The standard mentions the need to identify security constraints:
    - *A good understanding of the user and sponsor needs – which are not necessarily the same – is needed to ensure any issues are identified at the earliest possible stage.*
- Develop objectives

  JTEN objectives
    - *Overarching: To gain a better understanding of the potential utility of using the US Joint Training and Experimentation Network (JTEN) to link a simulation based in the US to one based in the UK.*
    - *Detailed: To allow a 'live' Forward Air Controller (FAC), out on the range at Westdown Camp, to direct a pilot in the US flying a simulated aircraft and for the resulting ground truth effects of any munitions dropped by that aircraft to be made visible – via Synthetic Wrap - to the FAC on the ground using an AR monocular to view the effects.*

This section mentions the need to identify:

- Security needs and constraints
    - *More than one level of security being used*
    - *Need to ensure no unauthorised release of data*
    - *How data and outputs will need to be stored – short term and long term*

- Potential security risks
    - *Identify by carrying out a risk assessment*
        - *Multiple nations participating*
        - *Possibility of unauthorised release of data (static and kinetic) to either users or networks*
        - *Risk of data leakage e.g. parameters for weapon or performance data*
        - *An aggregation of data may raise the classification levels*
        - *Deduction from the actions/reactions of participants may reveal classified information*
- Probable security level
    - *A combination of Secret (US), Secret (UK) and Restricted (UK)*
- Possible designated approval authority (or authorities, if a single individual is not possible)
    - *US and UK accreditation authorities:*
        - *Hardware*
        - *Software and data (e.g. terrain databases and 3D models)*
        - *Networks*
        - *Sites*
        - *People – although the individuals may not be known at this stage*
- Conduct initial planning
- As a potential outcome DSEEP lists:
    - *Security plan*
    - Sections where security is implicit:
- DSEEP recommends defining a high-level schedule of key development and execution events in section 4.1.3. This may include planning of security

Step 1 MSG-080 Suggestions:

- *Get accreditors involved!*
- *Handling of collective simulation between nations: Establish controlled processes and formal agreements (e.g. memorandum of understanding, MOU). These need to cover everything from the design phase to the data protection of after the exercise has finished.*

Step 2: Perform conceptual analysis

Develop simulation environment requirements

This section lists the tasks:

- Define security requirements for hardware, network, data, and software.
    - Networks must be accredited for the intended use
    - Software and data must be accredited for the intended use
    - Hardware must have passed evaluation to an agreed appropriate level
    - Measures are likely to be needed to manage the flow of data
    - Need to decide who will be allowed to see what

Step 2 MSG-080 Suggestions:

- *Add – we need an understanding of the impact on the training objectives of the security measures proposed. At this stage it may be necessary to review the training objectives and/or the security measures.*
- *Also need to understand the financial burden of implementing the security measures.*

Step 3: Design Simulation Environment

DSEEP Section 4.3.4 – Prepare detailed plan – suggests the following activity:

- Define security plan identifying needed simulation environment agreements and plans for securing these agreements.
    - *The live exercise, the AWES system and the FAC could not receive data from the simulations in the Secret enclave due to the use of an approved data diode; all data from the AWES live tracking system and simulation was passed into the Secret enclave.*
    - *JTEN used an air gap operator (controlled information flow) to transfer pre-agreed information from the Secret enclave to the Restricted enclave.*
        - *Potential for latency leading to discrepancies between the participating simulations*
        - *Potential for the introduction of errors by the 'Man-In-The-Loop'*
- The following outcome is also suggested:

- Security plan

Step 3 MSG-080 Suggestions:

- *Consider selecting federates in a way that minimizes the impact of the security classification.*
- *Review again the impact on the training objectives of the proposed security measures*
- *Each participant needs to identify the information security issues that are relevant to their assets: which type of information is releasable in what form or way and to which other participant(s).*
- *Decide in which ways the information will be released or could be released either intentionally (e.g. data exchange during runtime) or unintentionally (voice or data exchange during execution or debriefing).*

Step 4: Develop Simulation

DSEEP Section 4.4.2 – Establish simulation environment agreements – mentions:

- Agreements on […] and security procedures are all desirable to facilitate proper operation of the simulation environment.
- Additionally, simulation environments requiring the processing of classified data will generally require the establishment of a security agreement between the appropriate security authorities.
- It also lists the tasks:
  - Review security agreements, and establish security procedures.
  - Perform required system administration functions (establish user accounts, establish procedures for file backups, etc.).

Step 4 MSG-080 Suggestions:

- *Design the simulation to maximise the training value that can be obtained within the security constraints. In the case of JTEN an example of this was a decision for the enemy target vehicle to remain static in an attempt to mitigate the discrepancies caused by the different classification levels of the simulations.*
- *Check the security measures will not have any hitherto unforeseen impact on the training objectives.*

- DSEEP lists the outcome:
  - Established security procedures
- DSEEP Section 4.4.4 Implement simulation environment infrastructure mentions:
  - Confirm that the infrastructure adheres to the security plan.

Step 5: Integrate and Test Sim. Environment

This section mentions accreditation, probably related to Verification, Validation and Accreditation (VV&A) rather than security.

- *Carry out a final check on the impact of the security measures on the training objectives.*
- *Check that compliance with the security requirements has not invalidated the V&V of the event – will the training goals still be met? Is it a realistic environment?*

Step 6: Execute simulation

DSEEP section 4.6.1 – Execute simulation mentions:

- "When security restrictions apply, strict attention must be given to maintaining the security posture of the simulation environment during execution. A clear concept of operations, properly applied security measures, and strict configuration management will all facilitate this process. It is important to remember that authorization to operate is usually granted for a specific configuration of member applications. Any change to the member applications or composition of the simulation environment will certainly require a security review and may require some or all of the security certification tests to be redone."

The following task is mentioned:

- Confirm secure operation in accordance with certification and accreditation decisions and requirements.

Step 7: Analyze Data and Evaluate Results

- Manage the risk for information leakage during After Action Review for example the risk that comments by participants or instructors on the exercise events lead to unwanted information disclosure.
- Handle security considerations with regard to logged data that is not releasable.

- Handle security considerations w.r.t. archiving of relevant engineering and exercise data for possible future use or re-use.
- *Review impact of security measures on:*
  - *The security requirements – were they maintained?*
  - *The success of the training objectives – how well were they achieved?*
- *Possible changes identified for future events:*
  - *The DIS detonation PDU (as used in the JTEN trials) contains descriptors of the location and magnitude of a detonation, but no weapon or performance parameters are passed – neither data on the type of munition nor when it was released. On these grounds, the event accreditors gave a verbal indication that for future exercises of this type the detonation PDU might be transmitted into the Restricted enclave.*

Other comments:
There appears to be no mention made regarding the archiving of information. This has been added to Step 1 since early identification of any major issues arising is essential.

4.2 NLD – mapping MTMD (Maritime Theatre Missile Defence) to DSEEP

4.2.1 Use case Description

The Maritime Theatre Missile Defence (MTMD) Forum consists of nine nations (the United States, Canada, Australia, Germany, The Netherlands, United Kingdom, France, Spain and Italy), with the key focus on improving maritime coalition interoperability and capability in the area of missile defence - for example, improvements in the area of command and control, and tactical data links (TDL). Modelling and Simulation is used for testing, evaluating and assessing the performance of proposed interoperability improvements in an early stage of development. National simulation assets are connected in a (distributed) simulation environment to support this.

4.2.2 Use case mapping on MTMD

The following paragraphs demonstrate how this use case might be mapped against DSEEP – the DSEEP steps are given in normal font, the mapping in *italics*:

Step 1: Define Simulation Environment Objectives

- The objective of the simulation environment is to determine (through simulation) the performance of interoperability improvements, by making use of available national simulation models of the maritime platforms in the coalition force. The simulation models need to be representative for the national platforms. This almost automatically leads to the use of classified sensor, effector and TDL models. This need was recognized from the beginning and is reflected in the objectives of the analysis.
- *The required classification level needs to be stated from the beginning. Participants need to start preparations to work at this level (specifically lab accreditation and secure network communication).*

Step 2: Perform conceptual analysis

- *Although the information that is used in this step is partly classified, none of the results are classified. By keeping results unclassified the project team was able to perform their work in an unclassified working environment (e.g. using regular phones, mail exchange and collaboration sites). MOEs and MOPs are formulated in a generic way, the scenario does not hold any details on the national platforms or threats, and the simulation environment requirements are also stated in a generic way. Where specifics are needed, this is done via an anonymous reference to a classified document.*
- *For distributed teams, try to work in an unclassified environment for as long as possible e.g. stating scenarios and simulation environment requirements in an unclassified way, and only using classified data by anonymous reference.*

Step 3: Design Simulation Environment

This step involves the design and development of the simulation environment based on the requirements from the previous steps.

- *A similar approach is followed as in step 2. By keeping the component configuration separate from the component logic, it is possible the maintain components at a lower classification level. This approach enables the project team to do integration and test in an environment with a lower classification level, using unclassified component configurations. The design of the simulation environment itself is unclassified, by using available (open) standards for connecting simulation models, like RPR-FOM and L16 BOM. All models in the simulation environment are at an equal playing level, i.e. there is no information filtering between models.*

Step 4: Develop Simulation Environment

- Development or modification of components.
- *A similar approach is followed as in step 2. By keeping the component configuration separate from the component logic, it is possible the maintain components at a lower classification level. This approach enables the project team to do integration and test in an environment with a lower classification level, using unclassified component configurations. The design of the simulation environment itself is unclassified, by using available (open) standards for connecting simulation models, like RPR-FOM and L16 BOM. All models in the simulation environment are at an equal playing level, i.e. there is no information filtering between models.*

Step 5: Integrate and Test Sim. Environment

- For this step a process had to be devised to overcome several constraints and limitations such as:
- The federation contains sensitive information and as such classified data.
- The participating partners are located far from each other, in different time zones, making co-ordination of the federation members difficult.
- *All tasks are performed in an unclassified (co-located or VPN) environment, with the purpose to switch to a classified (co-located) environment for the final test event.*
- *With these constraints shown above and past experiences on (classified) network setup and performance, the following decisions were made at that beginning of simulation environment development:*
    - *A test federate shall be used to support local component interface and behaviour testing as much as possible.*
    - *The final simulation environment shall execute at one location to overcome distributed network delays and security issues.*
    - *In order to increase efficiency during co-located integration and test, existing (classified) components shall be modified or re-developed as configurable (unclassified) components.*
- *Another advantage of the introduction of unclassified components is the ability to conduct geographically distributed testing via an unclassified VPN connection between the project members.*

Step 6: Execute Simulation

- *The final test event is performed in a classified environment, where each of the models can be configured with classified data.*

Step 7: Analyze data and evaluate results

- *The data is collected and stored on removable hard disks, for analysis and evaluation in a classified environment.*

5. Way Forward

The JTEN use case shows how a security overlay to DSEEP would assist with the integration of security into the development process. Whilst a complete solution to all the issues is unlikely in the foreseeable future this does not mean no progress can be made. Raising the issue to the user and accreditor communities and providing a framework to adopt would be a step in the right direction.

At the Spring 2013 meeting of the SiS SSG a decision was made to draw up a Product Nomination for the creation of a security overlay to DSEEP, a Best Practice Guide to act as a reference point and for a glossary to ensure a common understanding of the terms used. As well as creating coherence in the development of an exercise, the intent is that these products will serve as a starting point for enterprise level engagement with the accreditation community, with the hope that this will lead to a better understanding of the issues and their impact on both sides.

This is a challenging topic and all SISO members are invited to join the SiS SSG and provide input to the product nomination.

The NATO Modelling and Simulation Group community will continue to support this activity and its members will be able to provide operational and technical experience with this problem. Experiments or exercises undertaken by NMSG task groups may serve as test cases for the proposed security overlay standard.

6. References

[1] J. A. Tufarolo, L. Suprise, M Raker, *International Interoperability for Simulation-based Training*, 97F-SIW-140, 1997

[2] C. A. A. Verkoelen, R. Wymenga, *Multi Level Security within Collective Mission Simulation Architectures*, 09S-SIW-035, 2009

[3] B. Möller, et al, *Towards Multi-Level Security for NATO Collective Mission Training – a White Paper*, 11S-SIW-069, 2011

[4] B. Möller, et. al, *Security in NATO Collective Mission Training - Problem Analysis and Solutions*, 12S-SIW-032, 2012

[5] B. Möller, S. Croom-Johnson, Wim Huiskamp, *Three Perspectives on DSEEP and Security: Training Goals, Use Cases and the Selection of Security Measures*, 13S-SIW-005, 2013

7. Acknowledgements

The authors would like to thank Ian Grieg of UK Defence Science and Technology Laboratory (Dstl) for his advice and review of the document and Tom van den Berg of TNO (NLD) for supplying the MTMD use case.

Author Biographies

STELLA CROOM-JOHNSON is a Principal Analyst in the Analysis, Experimentation and Simulation Group in the UK Defence Science and Technology Laboratory (Dstl). For the past few years she has acted as the technical lead on a UK MoD project looking at options for achieving a persistent Cross Domain Solution across standards and domains and has been a member of NATO Modelling and Simulation Group-080 (Security in Collective Mission Simulation). She is currently chair of the SISO Security in Simulation Standing Study Group.

WIM HUISKAMP is Chief Scientist Modelling, Simulation and Gaming in the M&S department at TNO Defence, Security and Safety in the Netherlands. Wim leads TNO's research programme on Live, Virtual and Constructive Simulation, which is carried out on behalf of the Dutch MOD. Wim is a member of the NATO Modelling and Simulation Group (NMSG) and acted as member and chairman in several NMSG Technical Working groups. He is co-chair of MSG-080, Chairman of the NMSG M&S Standards Subgroup (MS3) and he is the liaison of the NMSG to the Simulation Interoperability Standards Organisation.

BJÖRN MÖLLER is the vice president and co-founder of Pitch Technologies, the leading supplier of tools for HLA Evolved, 1516-2000 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 modelling and simulation, artificial intelligence and Web-based collaboration. He is currently serving as the vice chairman of the SISO HLA Evolved Product Support Group.

Developing the HLA Tutorial Part Two: Towards Federation Design Patterns

Björn Möller, Pitch Technologies, Sweden
Fredrik Antelius, Pitch Technologies, Sweden
Mikael Karlsson, Pitch Technologies, Sweden

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

Keywords:
HLA, Tutorial

ABSTRACT: *The first part of the HLA tutorial, based on HLA Evolved, released in 2012, was well received by both industry and academia. It is used, for example, for teaching distributed simulation in universities and also in the Smackdown project, originally initiated in SISO. The focus of the tutorial is to teach best practices for how to develop HLA federates and federations, rather than to cover all details of the HLA standard.*

The first part of the tutorial can be seen as the foundation for all federates, covering how to join a federation, publish, subscribe and work with objects and interactions. The recently released, second part, can be seen as a smorgasbord of techniques that can be used in federations, like time management, ownership, development of reusable FOM modules, fault tolerance and more. Few federations are likely to use all of the techniques but most federations will use some of them.

The biggest challenge is to explain the full potential of time management. The tutorial starts with a common and easily understood use case and gradually moves on to the general theory and more advanced time management topics.

One recurring theme of the tutorial is the best practices and design patterns typically used in federations. These are summarized in the end. Pointers are also provided to design patterns used in the NATO NETN design as well as in civilian HLA applications.

As a conclusion of the tutorial the concept of interoperability is revisited, based on the Layers of Conceptual Interoperability Model (LCIM).

1. Introduction

This paper focuses on part two of a freely available HLA Tutorial. The development and philosophy of the first part is described in a previous paper. Some background about why this effort was initiated and some of the challenges with getting started with HLA, using only the standards documents, is described in that paper.

1.1 Reactions to the HLA Tutorial Part One

The tutorial has been downloaded by thousand of readers. The authors have noted several interesting reactions to the first part of the tutorial.

Several people in the standards community have questioned if a tutorial is mainly a rewrite of the HLA specification. This may violate the copyright, in this case the IEEE copyright. A quick look at the tutorial reveals that it describes how to build federates and federations, not on the structure or details of the HLA specification. Not much detail is provided on each service call. Instead the tutorial points to the appropriate section of the HLA specification. If this was a carpentry tutorial, one might say that the focus is on how to build a house using the tools, not to provide a specification of a carpenters tools, material or building blocks.

The tutorial has been extensively used in the "Smackdown" University outreach project, originally initiated by SISO. The tutorial has been helpful in getting students up to speed with HLA. Since the first part does not cover HLA Time Management, students have used sample federates developed by NASA staff. Hopefully part two will help fill that gap.

Another interesting reaction is that several universities, for example in the UK, have adopted the tutorial as part of courses in distributed systems. This has happened without any direct involvement from the authors. A next step is probably to work more closely together with universities to develop additional lecture material.

Yet another interesting observation is that the authors have met many engineers from the simulation industry that maintain their own, neatly printed, copy of the tutorial. It is likely that a "print-on-demand" version of the HLA tutorial will be made available in the future.

2. Structure of the HLA Tutorial Part One and Part Two

Part one of the HLA tutorial consists of three main parts:

- An overview chapter that describes the origin and purpose of the HLA standard, the users of HLA as well as policy and market aspects.
- Two chapters that provides an overview of the architecture from a service-oriented perspective, where HLA can be considered a "Services Bus".
- Eight chapters that describe how to build a federation (federates and FOM), step by step. The structure of a FOM and federates are introduced, step-by-step.

In addition to this there are several appendices, most notably the Federation Agreement and the FOM. Source code and tools are also freely available for download for users that want to get s hands on experience or that need a starting point for their own development. Note that there is plenty of important information in the tutorial that is not provided at all in the HLA standard, for example how to structure a federate, how test and debug federates and the concept of object oriented HLA.

The second part of the HLA tutorial builds upon part one. It starts with four main sections:

- FOM modules and OMT data types
- Ownership
- Time Management
- Data distribution management

In addition to this there are several chapters with advice on interoperability and how to build HLA federations, including federation performance and fault tolerance. To promote a holistic view of federation development, the tutorial ends with a discussion around the Levels of Conceptual Interoperability Model (LCIM).

Part One of the tutorial is intended to be read in its entirety by a developer. Part Two can be read in any order, depending on the needs of the federation that the developer intends to develop.

3. More on the Main Sections

This section gives some additional insight in the main sections of Part Two of the tutorial.

3.1 FOM Modules

The FOM Modules section starts off with the monolithic FOM developed in Part One of the HLA tutorial. It then presents two main considerations when developing a FOM module:

- What is the purpose and scope of the FOM module
- What is the intended degree or reuse of the FOM module

This is then illustrated by splitting up the FOM into a general Federation Management module and a specific Fuel Economy FOM module. In most practical cases a FOM module needs some modifications to become generalized before it can be considered to be reusable. This is illustrated through a generalized scenario handling interaction.

One important FOM module that an HLA developer needs to understand is the predefined MIM module. It contains, among other things, some predefined building blocks for HLA Datatypes. How to build different types of Datatypes, such as Simple, Enumerated, Array and Record data types is described in detail.

3.2 Ownership Management

The principles of HLA ownership management are not obvious to many programmers with an object-oriented background. This section describes the purpose and typical use cases for ownership management. It also points out the implicit ownership that the creator of an object instance has. It also describes the importance of understanding that a distributed simulation needs to handle both locally created object instances as well as discovered, remotely registered object instances. The latter usually creates some confusion for developers that are used to develop code for simulations that receive little external data.

This section then moves on to describing some fundamental principles, "push" and "pull", of HLA ownership. The focus of the tutorial is "pull" ownership. In most practical applications, the HLA ownership services alone are not enough for managing ownership transfer. An example of a typical design pattern is provided; in this case a centrally managed "pull transfer of entire instance" is described.

The acquisition of ownership of attributes where the registering federate has been lost is also covered.

3.3 Time Management

This the most challenging part of HLA and it is presented in the longest chapter of the tutorial. It starts by presenting three important time concepts:

- Wall clock time
- Scenario time
- Logical time

A straightforward implementation of HLA Time Management in the Fuel Economy federation is then shown. A frame-based approach is used, meaning that all federates uses a fixed and equal time-step. In each "frame" the state of next "frame" is calculated. The cycle with granted/advancing state is shown together with the flow of outgoing and incoming time stamped events.

Simulation speed and pacing is then described and solutions are described for real-time, scaled real-time and as-fast-as-possible simulation.

Once the practical example is understood the theory of HLA Time Management is presented, introducing Look-ahead and Greatest Available Logical Time.

3.4 Data Distribution Management

This section shows how to use DDM with both dimensions that can be considered "continuous", like the car position, and "discrete" dimensions, such as type of fuel.

4. Discussion

4.1 What parts of HLA do we need?

Part One of the tutorial can be seen as the foundation of any HLA federation. The knowledge and the services described will be used by anyone that intends to build a federation.

Part Two of the tutorial provides a "smorgasbord" of HLA features. Most federations will use some of these features but few federations will use all of them

4.2 Towards design patterns

One recurring theme in the practical examples is that overarching distributed algorithms, or "design patterns" are usually developed, where HLA services can be considered building blocks. Some examples of the Fuel Economy Federation are the scenario management, the execution management and the management of how and when ownership transfer is performed. The NATO Education and Training Network groups (NMSG-068, NMSG-106) has also worked extensively with design patterns, for example for providing services, such as logistics and refueling, between simulated entities. The authors would like to argue that this is a trend that will continue. SISO should be one of the main forums for exchanging experiences from distributed design patterns for simulations.

5. Conclusion

The purpose of the HLA tutorial project is to increase the interest in distributed simulation in general and HLA in particular by lowering the barrier to the HLA standard. This is a long-term project, but the interest in the tutorial and the number of downloads already indicates success.

The next step is to collect feedback on the best practices presented and perform one more revision of both Part One and Part Two during 2014.

The authors also hope to see an increased activity around design patterns for distributed systems within SISO the coming years.

References

[1] "The HLA Tutorial", www.pitch.se, September 2012

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

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

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

[5] IEEE: "IEEE 1516-2010, High Level Architecture (HLA)", www.ieee.org, August 2010.

[6] Frederick Kuhl, Richard Weatherly, Judith Dahmann: "Creating Computer Simulation Systems: an Introduction to the High-Level Architecture", Prentice Hall PTR (2000), ISBN 0130225118

[7] DMSO: "RTI 1-3-Next Generation Programmer's Guide Version 3.2", September 2000, US Departement of Defense: Defense Modeling and Simulation Office

[8] "Porting a C++ Federate from HLA 1.3 to HLA 1516" & "Porting a Java Federate from HLA 1.3 to HLA 1516", March 2003, http://www.pitch.se/support/pitch-prti-1516

[9] "Migrating a Federate from HLA 1.3 to HLA Evolved", November 2010, http://www.pitch.se/technology/about-hla-evolved

[10] "FEAT PDG - Federation Engineering Agreement Template", www.sisostds.org

[11] SISO: "Real-time Platform Reference Federation Object Model 2.0 ", SISO-STD-001 SISO, draft 17, www.sisostds.org

Author Biographies

BJÖRN MÖLLER is the Vice President and co-founder of Pitch Technologies. 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 M.Sc. 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 Support Group and the chairman of the SISO Real-time Platform Reference FOM PDG.

FREDRIK ANTELIUS is a Senior Software Architect at Pitch and is a major contributor to several commercial HLA products, including Pitch Developer Studio, Pitch Recorder, Pitch Commander and Pitch Visual OMT. He holds an M.Sc. in Computer Science and Technology from Linköping University, Sweden.

MIKAEL KARLSSON is a is the Infrastructure 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.

Building Scalable Distributed Simulations: Design Patterns for HLA DDM

Björn Möller
bjorn.moller@pitch.se

Martin Johansson
martin.johansson@pitch.se

Fredrik Antelius
fredrik.antelius@pitch.se

Mikael Karlsson
mikael.karlsson@pitch.se

Pitch Technologies
Repslagaregatan 25
582 22 Linköping, Sweden

Keywords: Scalability, HLA, DDM, Design patterns

ABSTRACT: *Over the last decades the size of scenarios in distributed simulation has grown considerably, for example in defense training. There is also a demand for larger number of federates within exercises. This means that federation scalability is an area of growing importance. The developers of HLA foresaw this and introduced not only class-based filtering, but also the HLA Data Distribution Management (DDM) for instance filtering. This is a very general and flexible mechanism for filtering. The challenge for many beginners has been to understand DDM and to develop efficient designs.*

This paper presents some design patterns for DDM and discusses their pros and cons as well as implementation and efficiency. One design pattern is Uniform DDM where all attributes of an object class have the same DDM dimensions available. This makes the use of DDM much easier in federations. Design patterns for filtering based on static properties (like the fuel type of a vehicle) and dynamic properties (like the position of a vehicle) are then covered.

A number of best-practices are also discussed, for example FOM design, handling of objects going in and out of scope as well as the usefulness of advisories. Life cycle challenges, like how to mix federates with and without DDM support are covered.

Finally, some thoughts are given on the design of general and reusable DDM schemes. As an example a number of DDM schemes are proposed for the RPR FOM.

1. Introduction

During the last decade, there has been a growing demand for scalability in distributed simulations. Defense simulation scenarios have grown and become more complex, for example in international civilian-military exercises. The number of simultaneous platform trainers in the same federation is also growing. While early High-Level Architecture (HLA) [1] integrations focused on integrating existing monolithic simulations, today federations are developed in a more modular way, using a larger number of smaller components. And federations developers, like any other community, are always trying to push the envelope.

The lack of scalability, from a bandwidth and CPU perspective, was one of several reasons for developing HLA as a successor of Distributed Interactive Simulation (DIS) [2]. Today, an increasingly common architectural pattern for reusing existing DIS simulations is to create an HLA backbone to which islands of DIS simulations are connected.

1.1. Where are the bottlenecks?

When building large distributed simulations there are many factors that can limit the scalability. In practice, two of the most common are:

Network bandwidth limitations. While Gigabit networks are now common in many Local Area Networks, long distance links still have limited capabilities. Simple math shows that a one-megabit link cannot reasonably carry more than 1250 updates/second of 100 bytes (a common update size for updating entity positions).

CPU limitations. In a distributed simulation it is necessary both to produce data, and to receive and process data from other simulations. Many simulations have limited capability for processing incoming updates, in particular if this feature was added later, rather than in the original design of the system. This problem gets worse as the federation grows.

Consider ten simulations that send 1000 updates/second each. If every simulation subscribes to all of the shared information, they will thus receive 9000 updates/second. Now consider increasing the number of simulations to 100. They will now receive 90 000 updates/s while still only sending 1000 updates/s. The bottleneck for processing incoming data is usually CPU, although graphical sub-systems and databases may also be a constraint.

What program code that causes the CPU constraint is generally not very well understood. Many developers believe that, when bandwidth is abundant, the processing done by a communication framework, like an RTI, is extensive compared to the simulation model. In reality, very little CPU is used by the RTI to transfer information from the network to the receiving simulation. In the next step, for example when a new aircraft position is received, extensive processing may be needed for determining the relative position and angle of that aircraft and all other aircrafts.

Understanding how and when incoming updates are processes may be crucial for optimizing a federation. In some cases, “lazy” strategies may work well, like avoiding calculations until data is actually needed, or until all data for a particular time frame has been received.

1.2. General approaches for improving scalability

There are a number of general approaches for increased scalability. The most obvious one, and easy to implement, is to increase the available bandwidth and CPU resources. Another is to refactor and optimize the system code implementation. Optimized federation design and smart use of services for distribution of data will also increase performance and scalability by allowing infrastructure implementations to perform sender-side filtering and other dynamic optimizations during runtime.

For bandwidth limitations, there are also some common approaches, like compression. This can be handled by the network equipment, or by the sending and receiving CPU. In the latter case, some CPU processing is traded for increased bandwidth.

Bundling is another approach, where several messages are sent in one bundle. This reduces the impact of the networking overhead, since it takes less effort to send ten messages of 100 bytes bundled together as one single 1000-byte message, compared to sending them separately.

In some network topologies it is possible to replace networks hubs, where all local systems share the same bandwidth, with switches, where each combination of senders and receivers can use the full bandwidth.

Beyond these general approaches it is hard to achieve any optimizations without deeper insights into the information exchange, for example what information that is needed by each simulator, and what the characteristics of the simulation data are.

1.3. Add domain specific information for scalability

If more domain specific information is available for the filtering, better scalability may be achieved. The most obvious example is the publish/subscribe scheme used in HLA Declaration Management (DM). Each HLA federate subscribes to the object classes and attributes it is interested in. The RTI only delivers updates for a particular class and attributes to interested federates. The same scheme is also

used for interactions. This scheme improves scalability when different federates have different interests. If all federates subscribe to all classes, little optimization can be achieved.

To further optimize the information that is delivered to each federate, it may be desirable to deliver data only for a subset of the instances of a given class. An aircraft simulator may only be interested in other aircraft in the same geographical area. A command and control simulation may only require the positions of ground vehicles belonging to a certain force. If such criteria can be provided to the RTI, it is possible to reduce how much data that a federate needs to process and how much data that needs to be delivered over the network. There is a service group called Data Distribution Management (DDM) in HLA that provides this type of filtering. This paper seeks to describe how to use DDM in practical applications and discuss the optimal way to use it.

It shall also be mentioned that there are several other domain-specific approaches. One example is dynamic aggregation and de-aggregation. In this case we chose to describe a number of entities as an aggregate, for example a platoon, battalion or brigade. When required, for example during a particular phase of the scenario, the aggregate is de-aggregated into a larger number of entities. This assumes that not all aggregates need to be de-aggregated all of the time, in which case no additional scalability is gained. Aggregation and de-aggregation is extensively used in command and control exercises.

Another example is to use predictive techniques like dead-reckoning. A sender can avoid sending messages when dead-reckoned values, on the receiver side, will be close enough to the real value. This is used in the DIS and Real-Time Platform Reference FOM (RPR FOM) [3,4] standards for the exchanging spatial data for physical entities.

2. Overview of HLA DDM

This section contains a brief introduction to HLA DDM, that also forms a basis for the design patterns.

2.1. General principle

The HLA Data Distribution Management services enables developers of a federation to perform filtering on any data that they need. Figure 1 shows how DDM extends upon class-based subscriptions.

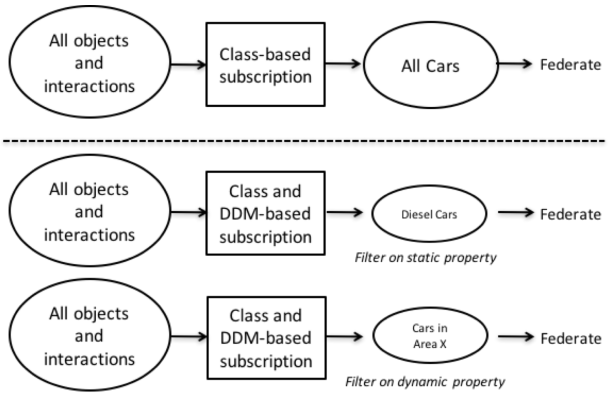


Figure 1: Subscription without and with DDM

If class-based subscriptions are used, a federate can choose to subscribe, for example, to all cars, but avoid to subscribe to aircrafts. When DDM is added, the federate can subscribe to diesel cars only, or cars in a selected geographical area. In the first case, filtering is done based on a static property of a car instance. In the second case, filtering is done based on a dynamic property of a car instance.

2.2. The normalization function

The key to understanding DDM is the Normalization Function. The purpose of the Normalization Function is to map any domain specific data in a federation into data with a generic format, in this case integer ranges, that the RTI can use. The RTI cannot reasonably be required to have any knowledge about a particular application domain. Detailed aspects like data types, enumerations, geospatial positioning need to be hidden. The usage of the Normalization function is shown in Figure 2.

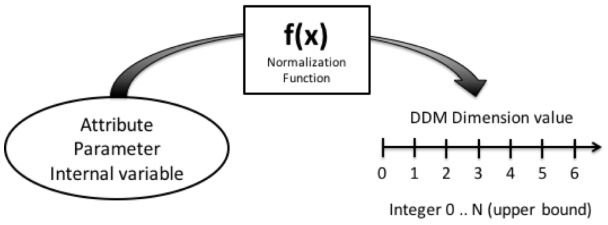


Figure 2: The Normalization function

The Normalization function takes input data, which could be attribute values, parameters, or any variable in a program, and converts it into an integer range in a user-defined Dimension. It is up to the developer to specify and implement a normalization function that meets his needs.

Consider the case of different types of fuel. The developer can introduce a Fuel Type Dimension. All types of fuel that

are used are then mapped into Ranges in this dimension, as shown in figure 3. When sending updates and interactions, or when subscribing, a DDM Region is used which specifies one or more Ranges, each one related to a Dimension.

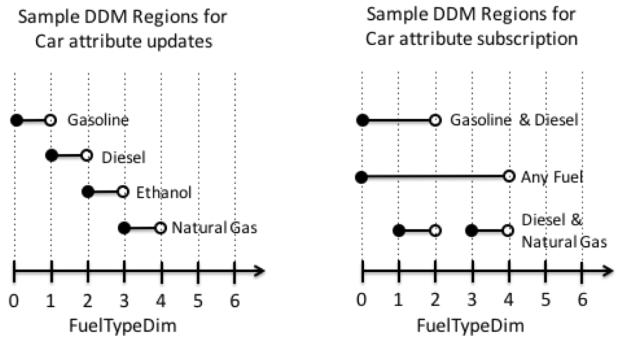


Figure 3: Sample Regions in the Fuel Type Dimension

In the left part of Figure 3 we can see that Gasoline is specified as the range [0..1), meaning that the range goes from 0 up to, but not including, 1. Diesel is specified as the Range [1..2). The value goes up to 4, which is the Dimension Upper Bound. The right part of Figure 3 shows regions that are used for subscribing, which will be covered in the next section.

2.3. Filtering at runtime

For each attribute and interaction class that needs to use DDM, the available Dimensions must be specified. Figure 4 shows how the Fuel Type Dimension is specified for one attribute of the Car class.

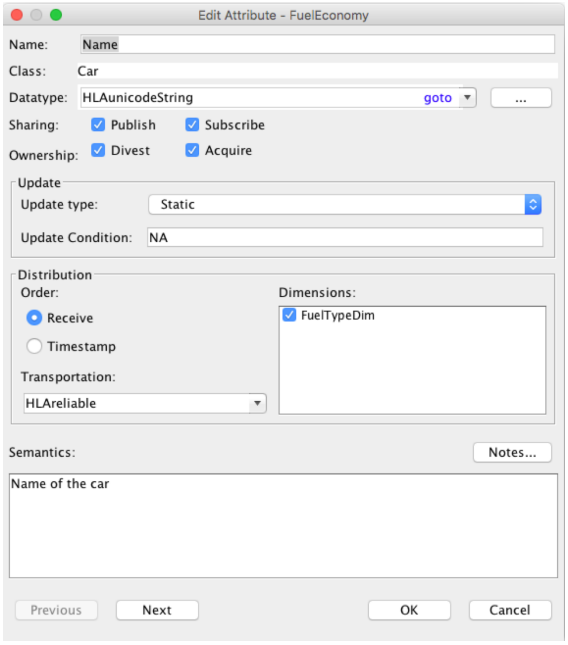


Figure 4: Specifying available dimensions

To perform filtering at runtime, Regions must be used as follows:

For the **federate that updates** an attribute (or sends an interaction), a Region shall be associated. As an example, when updating an attribute of a car, the Diesel Region (see Figure 3, left side) could be specified.

For the **federate that subscribes** to that attribute (or interaction), a subscription Region shall be provided. As an example, the Gasoline & Diesel Region (see Figure 3, right side) can be used.

The RTI will then compare these Regions when the update is sent. If the Regions of the update and the subscription **overlap**, then the update will be delivered.

To conclude, the DDM services enable federation developers to filter on any data that they have available. Any data can be used as input to Normalization Functions, which are used to determine Regions in one or more dimensions. The subscription requirements, expressed as Regions, are compared to the Regions of the updates or interactions.

2.4. Scope and Advisories

Consider a federation using DDM, where a federate that displays a map subscribes to gasoline & diesel cars. The federate then changes the subscription to gasoline cars only. This means that no more updates are received for diesel cars. This is known as the diesel cars going **out of scope**. All these cars will now freeze if displayed on a map display. In order to make it easier for the map display federate to handle this, for example by removing, or greying out these cars on the map, the RTI sends **out-of-scope** callbacks to the federate. Should they later come into scope, there are corresponding **in-scope** callbacks. These callbacks are called advisories. The updating federate gets the **turn-updates-on** and **turn-updates-off** advisories, to let it know if there are any federates that will receive any updates that it makes. In the above example, out-of-scope happens since the subscription region was changed. It may be just as common that an object and its attributes go out of scope since the updating federate changed the associated DDM regions.

2.5. Why isn't DDM more widely used?

DDM has proven very useful in some large federations. All major RTIs support it. Still, it is not extensively used. Some reasons for this may be:

- Many modest-size federations do not have a scalability problem.
- Federations with legacy simulations, where the source code may not be available, cannot implement DDM in

all participating federates. This limits the degree of filtering that can be made and thus the value of DDM.

- The DDM configuration – dimensions, ranges and normalization functions – needs to be agreed upon in the entire federation. This limits the opportunity to reuse a federate in a different federation, or at least requires coordination across organizations or standardization.
- It is not very clear from the HLA specification, or other documents, how to implement good DDM. The key to a good DDM design is the normalization function, which only has a short and formal description in the standard.
- The most commonly used reference FOM in the defense domain, the Real-time Platform Reference FOM (RPR FOM), does not provide any Dimensions or Normalization Functions. One explanation for this is that the RPR FOM maps to the older DIS standard, where no DDM is available.

The authors of this paper hope to make some improvements with respect to the two latter reasons.

3. Design Patterns for DDM

Design patterns are the re-usable form of a solution to a design problem. The idea was first introduced by the architect Christopher Alexander [5] for architecture and urban design. He describes a pattern as follows: "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." The concept of design patterns has since been used in many disciplines, not the least computer science, where it has made a big impact through the works of Erich Gamma et.al., also known as the "Gang of Four" [6].

3.1. Two general recommendations

Before looking at the design patterns, two recommendations are provided. The first and most important recommendation to DDM designers is to **keep the DDM design simple**. If just one simple Dimension with ten regions is used, and the data is evenly distributed across these regions, there is a potential of removing, on average, 90 percent of the incoming data for each federate, which means a big leap in scalability. This assumes that the most optimal Normalization Function is used for the particular domain and scenario.

A second, related recommendation, is to **avoid inventing "nice-to-have" DDM schemes** in a federation. Make sure that the design patterns truly divide the simulation data space into partitions that are meaningful to participating federates. Each additional DDM scheme that is required in a federation imposes some work on federate developers.

3.2. Pattern 1: Uniform DDM

Let's assume that we use different Dimensions and/or Normalization functions for each attribute of an object class. The result will be that different attributes will go in and out of scope at different times. This becomes very complicated to handle in a program. Parts of the remote object are up to date and parts are out of date. The parts that are up to date varies all of the time. A clearer design is achieved if all attributes go in and out of scope at the same time. The design pattern Uniform DDM is defined as follows:

"All attributes of a given object class shall have the same available Dimensions. Federates that update any of these attributes, shall provide regions for all Dimensions, using the Normalization Function associated with each attribute."

For the Fuel Type example this means that the entire Car instance, with all attributes, will be either in or out of scope.

3.3. Pattern 2: Static DDM

The Fuel Type pattern, described in the previous chapter, is a good example of the Static DDM pattern. There is a fixed set of regions along one Dimension, in this case regions for Gasoline, Diesel, Ethanol and Natural Gas along the Fuel Type Dimension. These regions are never modified. Each Car instance is associated with one region. This association never changes.

Another example is the Force Identifier in the RPR FOM. Platform instances can be associated with Regions connected to the Force Identifier Dimension. It will then be easy to subscribe to entities that are associated with the Friendly, but not the Opposing forces.

The Static DDM pattern, when applied to a class, is defined as follows:

"A fixed set of Regions with static Ranges are used. The object instance attributes are associated with the same Region throughout the federation execution."

This pattern is very efficient since there is no need for the RTI to recalculate the region overlaps after the initial calculation. The limitation is that the Normalization Function needs to be based on an input variable that is constant, like the fuel type of a car.

3.4. Pattern 3: Dynamic Checkerboard DDM

In many cases, we want to filter on geographical position. Since the position of a car is not expected to be constant during the federation execution, we cannot use the previous

design pattern. For best efficiency we still want to use a fixed set of Regions. We design it so that it creates a grid or a "checkerboard" across the map, as shown in Figure 5.

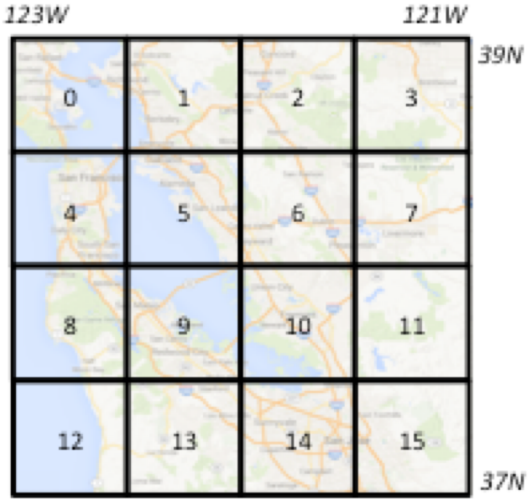


Figure 5: Checkerboard DDM regions

In this case we use a four by four grid and get sixteen Regions, from [0..1) to [15..16). Each Car instance is associated with a Region based on its position. A subscribing federate can select which Regions that it is interested in. Figure 6 shows how a car is associated with square 5. A federate has a subscribing region of square 0, 1, 4, 5 and will thus receive updates for that car.

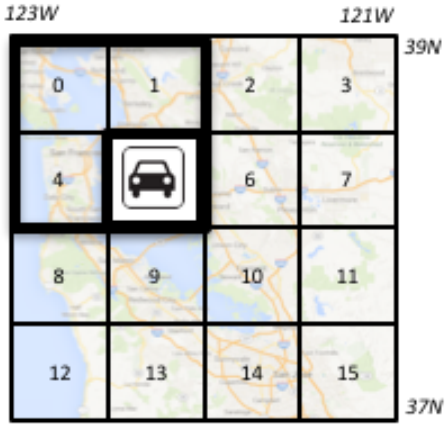


Figure 6: Checkerboard regions for updates and subscriptions

The Dynamic Checkerboard DDM pattern, when applied to a class is defined as:

"A fixed set of Regions with static Ranges are used. The object instance attributes are associated with one Region at a time. This association may change throughout the federation execution."

This patterns handles input variables that change over time, like the position of a car. Even with modest grid sizes, the filtering can be very powerful. A ten by ten grid can give an average update reduction of 99 percent. The recalculation of the Region overlap is usually very limited, and is caused by federates changing their subscriptions. You may also design other sets of static Regions, for example the States of USA.

The limitation is that it may be difficult to design a grid that fits any scenario. The example in Figure 5 has one square (number four) that contains San Francisco, which could be expected to contain considerably more cars than other squares.

3.5. Pattern 4: Dynamic Floating DDM

In order to make the filtering more exact than with the fixed checkerboard grid, we can tailor the region to each actual car. We will now introduce one Latitude Dimension and one Longitude Dimension. We may for example use a Normalization Function that maps a latitude range to a Latitude dimension with a Dimension Upper Bound of 100, i.e. values of [0..100). Each car is associated with its own Region that is constantly updated to match the position of the car. This is shown in Figure 6.

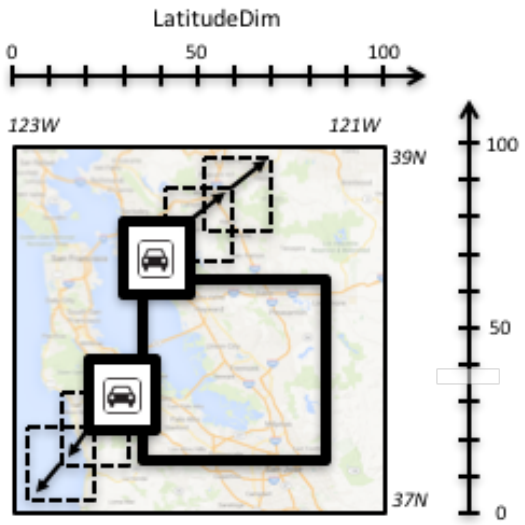


Figure 7: Dynamic floating regions

In the figure we can see two smaller Regions associated with two cars that move. There is also a larger Region, used by a subscribing federate. There is an overlap between the subscribing Regions and the two Regions associated with the cars. Note that the RTI has no insight in exactly where in the updating Region a car is located. If there is even the slightest overlap between the Regions of the updating federate and the subscriber, updates will be passed to the subscriber.

The Dynamic Floating DDM pattern, when applied to a class is defined as:

"One Region per object instance is used. The attributes of the object instance are associated with this Region. The Ranges of this Region may change throughout the federation execution".

Determining the size of the Regions used when updating the cars may be a challenge. They should be reasonably large, so that we do not need to modify them too often. At the same time, they should be kept small, in order to get higher accuracy when another federate subscribes to a region.

As a rule of thumb, consider the update rate for the cars and make sure that the regions are not updated more often than this, since changing the filtering conditions more often than sending data is suboptimal. However, the biggest limitation with this design pattern may be that it uses a lot of Regions where the Ranges are frequently recalculated. This may cause recalculation within the Local RTI Component across all federates.

3.6. Comparison

The following table summarizes the three design patterns.

Pattern	Instance Attributes	Regions
Static DDM	Statically associated with one region	Fixed set of regions with static, predefined ranges
Dynamic Checkerboard DDM	Associated to one region at a time. This association may change to other regions.	Fixed set of regions with static, predefined ranges
Dynamic Floating DDM	Each object instance is associated with its own region	One region per object instance. Ranges in the regions may change dynamically.

There are of course many other potential design patterns for DDM. The purpose of this section is to show a few proven patterns that federation developers can start with and apply as is.

It should also be noted that the discussion above builds on the assumption that it is computationally more expensive to modify regions (and thus recalculate region overlap) than to change the association of attributes to region.

4. Tool Support

This section shows how the implementation of DDM is supported in three commercial tools.

4.1. Developing FOMs with DDM

In Pitch Visual OMT, Dimensions are specified in a dedicated editor, as shown in the example in Figure 8.

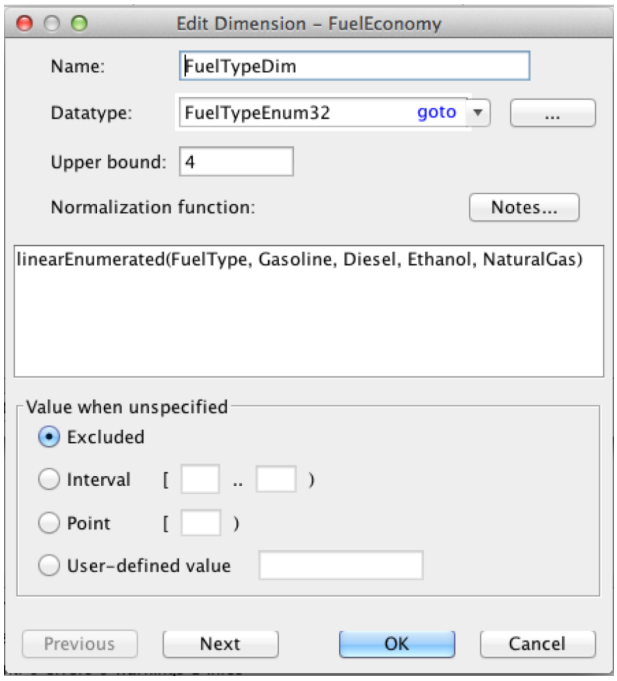


Figure 8: Defining a Dimension

When Dimensions have been specified they can be selected for attributes and interactions, as shown in the example in Figure 9.

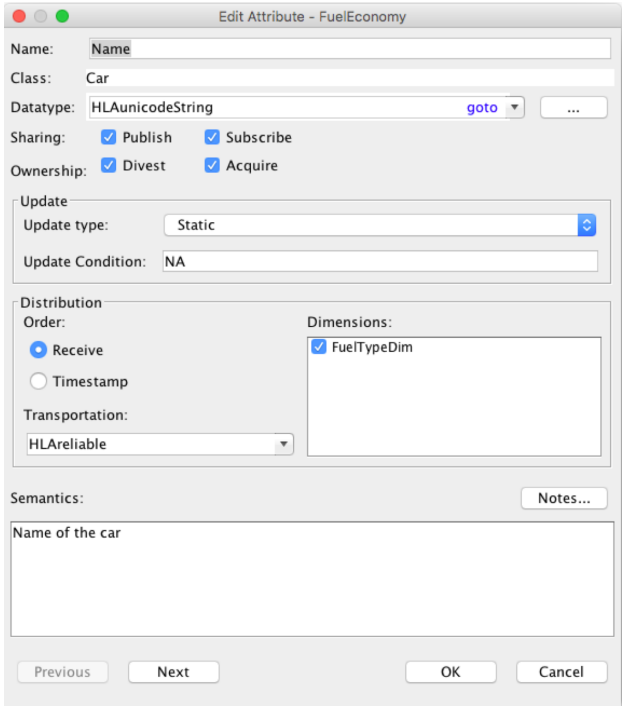


Figure 9: Specifying Dimensions for an Attribute

Pitch Visual OMT also provides an advanced tool that makes it possible to quickly specify Uniform DDM in large FOMs with many classes and attributes.

4.2. Federate Development in C++ and Java

Pitch Developer Studio generates C++ and Java code based on FOMs. It explicitly supports DDM based on the above design patterns.

4.3. Verifying and Debugging at Runtime

Pitch pRTI offers the ability to inspect and verify the DDM properties used in a federation. Figure 10 illustrates how to inspect the Region associations of Car attributes.

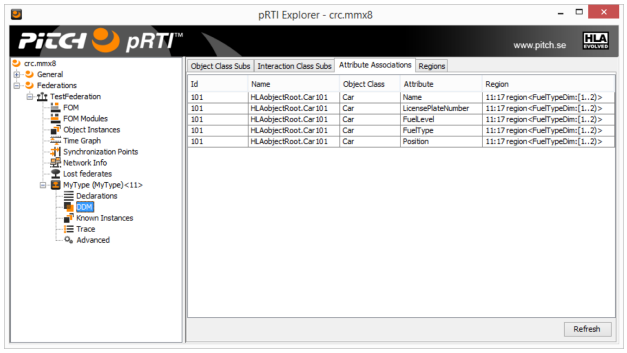


Figure 10: Region to Attribute inspection in Pitch pRTI

There are also views to inspect the Ranges of all Regions as well as the Regions used by subscribing federates.

5. DDM for RPR FOM

A standard scheme for RPR FOM would be highly desirable to facilitate the development of large federations in the defense and security domain. As previously mentioned, DDM and Normalization Functions should be kept simple. Based on discussions around the RPR FOM and the extension in the NATO Education and Training Network (NETN) design, the following four DDM approaches would be the most helpful ones for many federations. Note that it should be possible to freely combine them.

Position using dimensions for Latitude and Longitude. The most obvious filtering criteria for many simulations is to get information only from a selected part of the battlefield.

Force Identifier. It is very common for a simulation to only process information about platforms for selected forces.

Domain (Air/Ground/Sea). This is another commonly used discriminator. It is particularly interesting when used together with the two previous approaches.

Echelon. This is important in particular in command and control applications.

6. Challenges and Considerations

This section discusses some challenges when implementing DDM as well as some particular aspects that need to be considered.

6.1. Mixing federates that use and don't use DDM

There may be challenges when building federations where not all federates use DDM. The challenge for federates that use DDM is that federates that do not use DDM will send data without associated Regions. This data will be sent in the Default Region, which is a region that matches all other regions. Federates that subscribe using DDM will still need to handle incoming data that would otherwise be filtered, thereby reducing the performance advantage that DDM would normally provide. A workaround can be to add Regions for older federates using RTI plug-ins or using a federation-to-federation bridge.

Federates that subscribe without using DDM will receive all data that is sent, regardless of any regions used by the sending federates. The challenge for these subscribing federates is that they may not able to process all the incoming data in a federation where DDM is assumed since they don't take advantage of the load-reducing filtering that DDM-enabled federates do.

6.2. Designing reusable Normalization Functions

When implementing DDM in a federate it is an advantage if it can be reused in different federations. Consider design pattern 3 and 4 in this paper. As described above, they are hard-coded to a particular geographical area. A better idea is to make them configurable, or to come up with a generic Normalization Function.

6.3. Understand the scenario

In many cases, the optimal Normalization Function depends on the scenario. Consider a scenario using checkerboard DDM for a large number of entities in a large geographical area. Now consider a similar scenario in a small geographical area. If we need to filter out five percent of the entities, then the size of the grid needs to be different in these two cases. Two possible solutions are to either use variable parameters for the grid resolution, or to have fixed, fine grained update regions, and vary the size of the subscribing regions.

Similar cases may occur for other types of Normalization Functions. In some scenarios, subscribing federates may only need to distinguish between entities in the air, land and sea domain, other scenarios need to distinguish between friendly, neutral and opposing entities. For some scenarios, the platform type may even be of interest.

Federation developers need to strike a balance between these requirements and the recommendation earlier in this paper, to avoid inventing "nice-to-have" DDM schemes.

6.4. Know your RTI

Another challenge is that different RTIs implement DDM in different ways. This paper will not go into this topic, since it would require a paper on its own. Developers are encouraged to study the specifics and configuration options of RTIs they are considering.

6.5. Towards cutting-edge scalability

This paper focuses on a set of good design patterns for getting started with DDM. Advanced DDM scheme developers may consider studying work done in JLVC and NCTE [7] that implements a world-wide DDM grid using both geographical and other dimensions, resulting in 55 million regions.

7. Conclusions

The size of simulation scenarios has grown considerably over the years. DDM is one of the most important tools for achieving scalability. This paper presents some experiences, some advice and four design patterns as a starting point for developers that want to start exploring DDM.

References

[1] IEEE: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)", IEEE Std 1516-2010, IEEE Std 1516.1-2010, and IEEE Std 1516.2-2010, www.ieee.org, August 2010.

[2] IEEE: "IEEE Standard for Distributed Interactive Simulations", IEEE Std 1278.1-2012, www.ieee.org, December 2012

[3] SISO: "SISO-STD-001.1-2015, Standard for Real-time Platform Reference Federation Object Model (RPR FOM)", www.sisostds.org, September 2015.

[4] Björn Möller et al.: "RPR FOM 2.0: A Federation Object Model for Defense Simulations", 2014 Fall Simulation Interoperability Workshop, (paper 14F-SIW-039), Orlando, FL, 2014.

[5] Alexander, Christopher (1977). "A Pattern Language: Towns, Buildings, Construction". Oxford University Press. ISBN 0-19-501919-9.

[6] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley. ISBN 0-201-63361-2.

[7] Andy Ceranowicz et al: "Revisiting Interest Management", 2014 Fall Simulation Interoperability Workshop, (paper 14F-SIW-041), Orlando, FL, 2014.

Author Biographies

BJÖRN MÖLLER is the Vice President and co-founder of Pitch Technologies. He leads the development of Pitch's products. He has more than twenty-five 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 a M.Sc. in Computer Science and Technology after studies at Linköping University, Sweden, and Imperial College, London. He is currently serving as the chairman of the Space FOM Product Development group and the vice chairman of the SISO HLA Evolved Product Development Group. He was recently the chairman of the SISO RPR FOM Product Development Group.

FREDRIK ANTELIUS is a Senior Software Architect at Pitch and is a major contributor to several commercial HLA products, including Pitch Developer Studio, Pitch Recorder, Pitch Commander and Pitch Visual OMT. He holds an M.Sc. in Computer Science and Technology from Linköping University, Sweden.

MARTIN JOHANSSON is Systems Developer at Pitch Technologies and is a major contributor to several commercial HLA products such as Pitch Developer Studio and Pitch Visual OMT 2.0. He studied computer science and technology at Linköping University, Sweden.

MIKAEL KARLSSON is the Infrastructure 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.

Die High Level Architecture: Anforderungen an interoperable und wiederverwendbare Simulationen am Beispiel von Verkehrs- und Infrastruktursimulationen

Ulrich Klein / Steffen Straßburger
Universität Magdeburg, Institut für Simulation und Graphik
Postfach 41 20
D-39016 Magdeburg
Email: {uklein@isg, strassbu@sunpool}.cs.uni-magdeburg.de

Der Beitrag stellt die High Level Architecture vor, die die Erstellung interoperabler und wiederverwendbarer Simulationsmodelle unterstützt. An einem Verkehrsbeispiel wird eine zivile Nutzung der aus dem militärischen Bereich stammenden HLA beschrieben und die Anforderungen aufgezeigt, die bei der Nutzung von HLA an klassische Simulationswerkzeuge gestellt werden.

1 Einführung

Mit dem Erscheinen der neuen High Level Architecture (HLA) des amerikanischen Defense Modeling & Simulation Office (DMSO) erhält das Gebiet der verteilten Simulation neue Impulse [1]. Die Arbeit beschäftigt sich mit der flexiblen verteilten Modellierung von Simulationen und allgemeineren Systemen sowie möglichen neuen Anwendungsfeldern, die durch HLA für Simulationen erschlossen werden können. Anhand des Beispiels einer Verkehrssimulation wird dargelegt, wie eine nach beteiligten Teilsystemen modularisierte Simulation aus einem herkömmlichen monolithischen Modell erstellt werden kann. Dabei werden die Anforderungen, die HLA an konventionelle Simulationstools stellt, beschrieben und derzeitige Grenzen aufgezeigt.

2 Die High Level Architecture (HLA)

Die HLA bietet einen Ansatz, die Forderungen nach Wiederverwendbarkeit und Interoperabilität von Simulationsmodellen umzusetzen und die Schwierigkeiten mit monolithischen Modellen bei Änderungen der Funktionalität oder Konnektivität zu umgehen.

Ähnlich dem CORBA zugrundeliegenden Prinzip wird eine einheitliche Schnittstelle (*HLA Interface*) definiert, welche die Teilnehmer eines gemeinsamen Simulationslaufs (*Federates*) aufzuweisen haben, um in Kontakt mit der *Runtime Infrastructure* (RTI) zu treten, welche Basis-, Koordinations- und Kommunikationsdienste zur Laufzeit bereitstellt. Eine *Federation* kann als ein Vertrag zur Durchführung eines Simulationslaufes (*Federation Execution*) zwischen den Federates angesehen werden, in dem die Einzelheiten und Objektmodelle der Federates (*Simulation Object Model*) und der Federation (*Federation Object Model*) festgelegt sind. Diese Informationen sind in einer definierten

Form zu dokumentieren (*Object Model Template*). Weiterhin legt HLA zwingende Verhaltensregeln für Federates und Federations fest. Ein Vorteil der Architektur ist die Offenheit für andere Arten von Federates; sowohl Softwarebausteine wie z.B. Datenbanken und Beobachter als auch reale Elemente wie Sensoren und andere Hardware können bei Einhaltung der HLA-Spielregeln bei Federation Executions mitwirken (Bild 1).

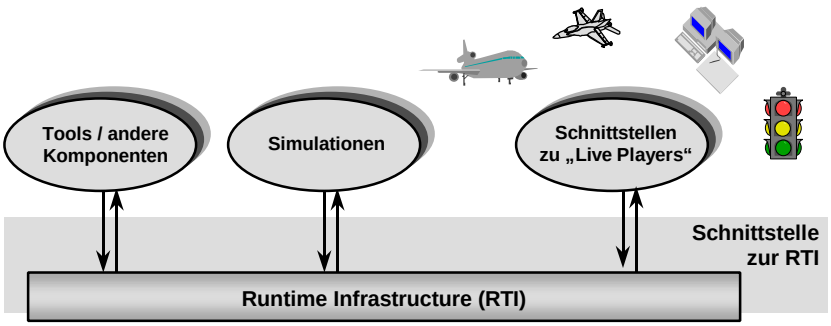


Bild 1 Aufbau der High Level Architecture

Ebenso wird ein transparentes Zeitmanagement unterstützt, welches den Federates außer der Koordinierbarkeit keine Vorschriften bzgl. des lokalen Zeitfortschritts (z.B. Zeitschritt-, Ereignis-, Echtzeitsimulation) macht.

3 Anwendungsbeispiel Verkehrs- und Infrastrukturmanagement

Unter Anwendung der HLA ist es möglich, ein System z.B. in Analogie zu dessen Subsystemen zu modellieren. Ein Beispiel für komplex vernetzte Subsysteme ist der Verkehr mit den zugrundeliegenden Infrastrukturen [4].

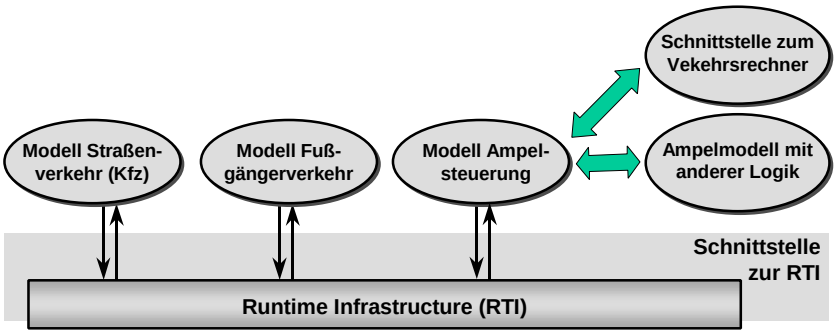


Bild 2 Aufbau des HLA-Beispiels einer Verkehrsregelung

So lassen sich bereits am Beispiel eines lichtsignalgesteuerten Verkehrsknotens Vorzüge der HLA zeigen (Bild 2): hierzu wurden jeweils getrennte Module geschaffen, die separat

den komplexen Verkehrsfluß von Autos, Radfahrern und Fußgängern modellieren. Eine weitere Komponente (Federate) enthält das Modell der Lichtsignalanlage (LSA) mit einer gewissen Steuerungslogik.

Durch die Flexibilität der High Level Architecture ist es möglich, dieses Ampel-Federate (aus Sicht der anderen Federates eine Black Box mit einheitlicher Schnittstelle und bekanntem Objektmodell) durch ein beliebiges anderes, z.B. mit einer anderen Steuerungslogik oder mit einer Schnittstelle zu einem Verkehrsrechner, auszutauschen.

4 Anforderungen der HLA an beteiligte Simulationen

Die oben erwähnten Vorzüge und Vorteile der HLA stellen teilweise beachtliche Anforderungen an die potentiellen Federates, die nachfolgend aus konzeptioneller und programmierungstechnischer Sicht beschrieben werden.

4.1 Konzeptionelle Anforderungen

In der verteilten Simulation können voneinander abhängige Simulationen ihren lokalen Zeitfortschritt nicht mehr autonom durchführen; vielmehr ist dieser (bei HLA mittels der RTI) zu koordinieren [2, 3]. So ist es z.B. bei ereignisgesteuerten Simulatoren nötig, die Ereignisketten von sich am Zeitfortschritt beteiligenden Federates miteinander zu synchronisieren bzw. verknüpfen. Es muß in einer gewissen Weise sogar die Möglichkeit bestehen, externe Ereignisse, die im lokalen Federate von Bedeutung sind, in die lokale Ereigniskette aufzunehmen. Dies ist z.B. nötig, wenn sich Attributswerte im externen Federate ändern, die im lokalen Federate von Interesse sind.

Eine weitere konzeptionelle Anforderung betrifft den Datenaustausch zwischen Federates sowie die Notwendigkeit der Darstellung externer Daten (Objekte, Attributswerte) im Federate (*ghosten*). Das HLA Konzept sieht vor, daß ein Federate für sich interessante Daten, die durch andere Federates modelliert werden, „abonniert“ und in der Folgezeit über Änderungen bezüglich dieser Daten informiert wird und diese ggf. intern abbildet.

4.2 Programmierungstechnische Anforderungen

Die Zusammenarbeit von Federates mit der RTI basiert auf einem Botschafterprinzip: dabei entsenden sowohl die RTI als auch das Federate ein Botschafter-Objekt zum jeweils anderen Kommunikationspartner; der RTI-Botschafter beim Federate nimmt Anfragen (Methodenaufrufe) entgegen, während die von der RTI gerufenen Dienste des Federate dem Federate-Ambassador beim RTI übergeben werden.

Die RTI-Software besteht (in der F.0-Version) aus dem RTIExec-Prozeß (der mehrere Federation Executions verwalten kann) sowie dem FedEx-Prozeß (der individuell für jede Federation Execution gestartet wird und dann die Kommunikation mit den Federates übernimmt). Außerdem muß ein potentielles Federate in der Lage sein, eine bereitgestellte RTI-Bibliothek einzubinden, welche das RTI-Ambassador-Objekt sowie das abstrakte Federate-Ambassador-Objekt enthält, wobei letzterer noch auszuimplementieren ist. Zur Zeit geschieht dies in C++; die Unterstützung von CORBA und weiteren Sprachen wie Ada95 und Java ist angekündigt.

Unsere Untersuchungen zeigten, daß es weniger die konzeptionellen, sondern vielmehr die programmierungstechnischen Forderungen sind, die potentielle Federates vor schwierige Aufgaben stellen.

5 Realisierung der Anbindung klassischer Simulatoren an HLA

Auf der Grundlage der oben genannten Anforderungen wurde stufenweise ein verteiltes Modell geschaffen. Auf dem Weg vom klassischen monolithischen Modell zu einem HLA-konformen verteilten Modell wurden in Zwischenstufen die Schnittstellen zwischen den Komponenten herausgearbeitet und zunächst mit einer nicht HLA-konformen Verteilung experimentiert. Das Modell beinhaltet die drei in Bild 2 dargestellten Basiskomponenten, die in der Endstufe als eigenständige Federates agieren. Für die Untersuchungen verwendeten wir die Simulationstools GPSS/H und SLX.

5.1 Zeitliche Synchronisation

In unserem Beispiel wurde sowohl eine zeitschritt- als auch eine ereignisgesteuerte Variante erstellt; eine Echtzeitkomponente ist vorgesehen. Unsere Lösung sieht das Einführen einer Planungsroutine bzw. eines Scheduling-Threads vor, der nebenläufig zur eigentlichen Simulation im Simulator abläuft. In der vorgestellten Lösung für GPSS/H (und für SLX analog) wird dies durch einen priorisierten Thread realisiert. Die zu wählende Priorisierung hängt von verschiedenen Interna des Simulators ab; insbesondere davon, wann Ereignisse in die zukünftige Ereigniskette eingetragen werden und wie gleichzeitige Ereignisse behandelt werden.

Das Arbeitsprinzip dieser Planungsroutinen besteht darin, das jeweils nächste Ereignis aus der zukünftigen Ereignisliste des lokalen Federates zu bestimmen (bei der zeitschrittgesteuerten Variante den nächsten Zeitschritt), den entsprechenden Zeitfortschritt bei der RTI zu beantragen und bis zur Antwort den lokalen Zeitfortschritt zu blockieren. Die Gewährung des Antrages (*TimeAdvanceGrant*) obliegt der RTI, welche die Abhängigkeiten der Federates und die verwendeten Synchronisationsprotokolle (optimistisch, konservativ mit individuellen dynamischen Lookaheads, etc.) berücksichtigt.

Die RTI teilt daraufhin mit, ob ein zeitlich früher gelegenes Ereignis (z.B. eine Änderung eines „fremden“ Objektattributs) zu berücksichtigen ist oder der beantragte Zeitraum gewährt werden kann. Nun kann die Planungsroutine zum gewährten Zeitpunkt fortschreiten und ggf. relevante Attributsänderungen durchführen. Bis zum nächsten Fortschritt der lokalen Zeit übernimmt das eigentliche Simulationsmodell die Kontrolle.

5.2 Datendarstellung und -austausch

Die Realisierung der Darstellung von externen Daten im lokalen Federate bereitete keinem der untersuchten Simulatoren größere Probleme. Es sei hier allerdings anzumerken, daß bisher nur statische Objekte untersucht wurden (logische Schalter, Variablen, etc.). Eine Lösung für die Darstellung von Informationen über dynamische Objekte (z.B. der Transaktionen in GPSS/H) ist Gegenstand laufender Untersuchungen.

5.3 Programmierungstechnische Umsetzung der Anforderungen

Für GPSS/H (Version für MS-DOS) ist es gelungen, sämtliche der oben erwähnten konzeptionellen Forderungen zu erfüllen. Eine direkte Anbindung an die RTI über die für die Intel-Plattform vorliegende Dynamic Link Library (DLL) war jedoch durch die beschränkten Kommunikationsmöglichkeiten von GPSS/H nicht realisierbar, da die GPSS/H-Funktionen zum Einbinden externen C-Codes keine Nutzung bereits compilierter DLL's zulassen. Eine GPSS/H-kompatible javabasierte Implementation mit HLA-Schnittstelle, die die o.g. Einschränkungen aufhebt, ist in Vorbereitung [5].

In der GPSS/H Implementation existieren 3 getrennte Module, die als an HLA angelehnte Federates agieren, sich jedoch nicht an die Interface Spezifikation halten und das RTI nicht nutzen. Sie synchronisieren ihre lokalen Zeiten unter Nachbildung der RTI-Funktionalität und eines konservativen Synchronisationsprotokolls unter Zuhilfenahme der oben erwähnten Planungsroutine. Aufgrund der eingeschränkten Kommunikationsmöglichkeiten von GPSS/H erfolgt der Datenaustausch über Dateien, womit die zweite konzeptionelle Forderung nur unter Umgehung der HLA-Interface-Spezifikation erfüllt werden konnte. Dies stellt nach unseren Erkenntnissen auch das grundsätzliche Problem bei vielen klassischen Simulatoren dar.

Eine Möglichkeit, Simulatoren trotz der oben aufgeführten prinzipiellen Beschränkungen an die RTI anzubinden, besteht in der Verwendung von Gateway-Programmen. Die Kommunikation zwischen dem Gateway-Programm, welches in C++ geschrieben werden könnte, und dem Simulator würde über Dateien bzw. Pipes erfolgen. Potentielle Funktionsaufrufe des Simulators würden in entsprechende Einträge in Dateien umgesetzt, vom Gatewayprogramm entsprechend ausgewertet und an die RTI weitergeleitet. Ebenso würden Funktionsaufrufe der RTI an den Simulator bei dem Gatewayprogramm erfolgen, welches dem Simulator die Information über Dateieinträge übermittelt. Dieses Konzept hat jedoch mehr theoretischen Charakter, da die Effizienz aufgrund des Pollings der Dateien bzw. Pipes nach neuen Daten sehr unbefriedigend wäre.

Erst bei der Untersuchung von SLX, eines relativ neuen Simulationstools für die Windows 95 / NT Betriebssysteme, gelang es, sämtliche Forderungen zu erfüllen und SLX als vollwertiges Federate zu verwenden. Es kamen das gleiche Modell und die gleichen Lösungsansätze für die konzeptionellen Anforderungen wie in der GPSS/H Implementation zur Anwendung, wobei wir in diesem Fall die Fähigkeit von SLX zur Einbindung externer DLLs nutzen konnten. Da die Aufrufkonventionen jedoch von SLX limitiert werden, wurde eine Wrapper-DLL entwickelt, die zwischen Aufrufen von SLX und eigentlichen RTI-Aufrufen vermittelt und gleichzeitig die Funktion des Federate Ambassadors übernimmt. Diese DLL löst auch gleichzeitig das Problem, daß man durch das C++ API an diese Sprache gebunden ist, und kann gleichzeitig als Basis für die RTI-Ankopplung von anderen Simulatoren dienen, die DLLs einbinden können.

6 Anforderungen für weitergehende HLA-Techniken

Beabsichtigt man die Nutzung von fortgeschrittenen Techniken der HLA (z.B. das Speichern und Laden des Zustands der Federation Execution), ergeben sich weitere, hier noch nicht diskutierte Probleme. Die entsprechende Funktionalität wird erst in späteren Versio-

nen der RTI implementiert, weshalb sich weitergehende praktische Untersuchungen verzögern. Diese zusätzlichen Anforderungen werden stärker als die oben diskutierten von der Unterstützung im konkreten Simulator abhängen. So ist z.B. in GPSS/H das Abspeichern und Wiederherstellen des Modellzustandes prinzipiell möglich (*Checkpoint*-Befehl.

7 Zusammenfassung und Ausblick

Es wurde gezeigt, daß es bei Erfüllung der hier vorgestellten Anforderungen möglich ist, konventionelle stand-alone Simulatoren als HLA-konforme Federates zu nutzen, wobei die Effizienz der vorgestellten Methoden beträchtlich differiert. Auch sind gewisse „Kosten“ wie z.B. für Koordination und Kommunikation, die für die Vorzüge wie Modularität und Wiederverwendbarkeit anfallen, zu berücksichtigen. Wurde für ein Simulationstool der Initialaufwand zur Erstellung einer RTI-Schnittstelle erbracht, so reduziert sich der eigentliche, HLA-bedingte Aufwand auf die Implementation der Modellkomponenten in den einzelnen Federates und die Aufstellung der Objektmodelle. An der Erstellung einer solchen allgemeinen Lösung für den Simulator SLX wird derzeit gearbeitet. Generell ist zu erwarten, daß die HLA aufgrund ihrer Flexibilität neue Anwendungsfelder, -techniken und -qualitäten erschließen kann; für klassische Simulationstools erlaubt dies einen vielversprechenden Schritt in Richtung wiederverwendbarer und interoperabler Simulationsmodelle.

Literatur

[1] Defense Modeling and Simulation Office (DMSO). *High Level Architecture Homepage.* URL http://www.dmso.mil/projects/hla

[2] Mehl, H. *Methoden verteilter Simulation.* Vieweg Verlag, Braunschweig, 1994

[3] Fujimoto, R. *Parallel Discrete Event Simulation.* In Communications of the ACM, 1990, no. 10, pp. 30-53

[4] Klein, U. *Zivile Anwendungspotentiale der High Level Architecture (HLA).* Symposium Neue Technologien in der wehrtechnischen Simulation, 30.09.-02.10.1997, Mannheim, in Vorbereitung

[5] Klein, U., S. Strassburger, J. Beikirch. *Distributed Simulation with JavaGPSS based on the High Level Architecture*. International Conference on Web-based Modeling and Simulation, Jan. 11.-14. 1998, San Diego, in Vorbereitung