



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

Copyright Pitch Technologies AB, Sweden, 2012. All rights reserved with the following exceptions: This document may be copied and redistributed for commercial and non-commercial purposes with the following restrictions:

- The document must be provided in its entirety. No part of the document may be removed or modified or reused in other documents. No content may be added.
- The copyright notice on the document cover and on each page must be included and clearly visible.
- It may not be translated without our prior consent.
- No charge may apply for the document, except for direct media costs.
- Copies may only be provided as PDF files or in print.

The document is provided “as is”. Pitch assumes no responsibility or liability for the content.

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
Åsa Wihlborg
Filip Klasson

Mikael Karlsson
Steve Eriksson
Patrik Svensson

Björn Löfstrand
Martin Johansson
Pär Aktanius

Table of Contents

1. Introduction	5
2. The Architecture and Topology of HLA	8
3. HLA Services	13
4. Connecting to a Federation	17
5. Interactions – Creating the FOM.....	24
6. Interactions – Calls for Sending and Receiving.....	30
7. Objects – Creating the FOM	35
8. Objects – Calls for Registering and Discovering	42
9. Objects - Calls for Updating and Reflecting Attribute Values.....	48
10. Testing and Debugging Federates and Federations	53
11. Object Oriented HLA	59
12. Summary and Road Ahead.....	64
 Appendix A: The Fuel Economy Federation Agreement.....	 67
Appendix B: The Fuel Economy FOM.....	74
Appendix C: Description of Federates and File Formats	78
Appendix D: Lab Instructions	83
Lab D-1: A first test run of the federation.....	84
Lab D-2: Connect, Create and Join.	86
Lab D-3: Developing a FOM for Interactions.....	88
Lab D-4: Sending and receiving interactions.....	90
Lab D-5: Developing a FOM for object classes.....	92
Lab D-6: Registering and discovering object instances	93
Lab D-7: Updating objects	95
Lab D-8: Running a Distributed Federation.....	97
Appendix E: The Sushi Restaurant federation	99
Appendix F: A summary of the HLA Rules	101

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:

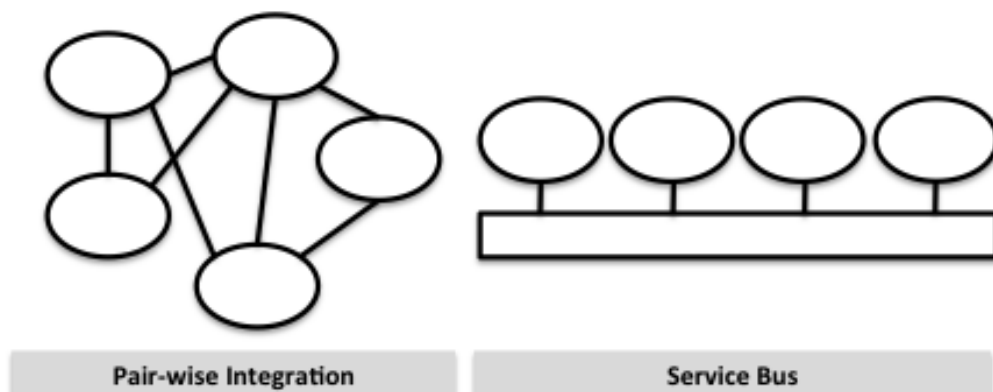


Figure 2-1: Two topologies

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:

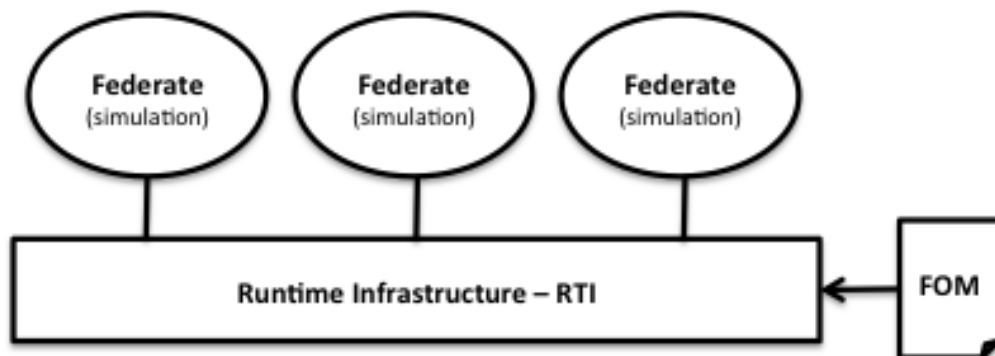


Figure 2-2: The topology of HLA

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.

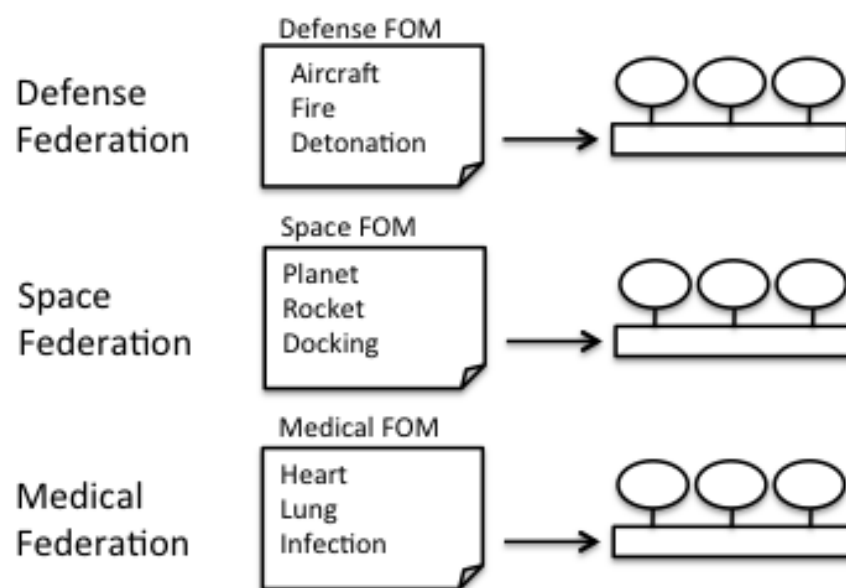


Figure 2-3: Different FOMs for different domains

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.

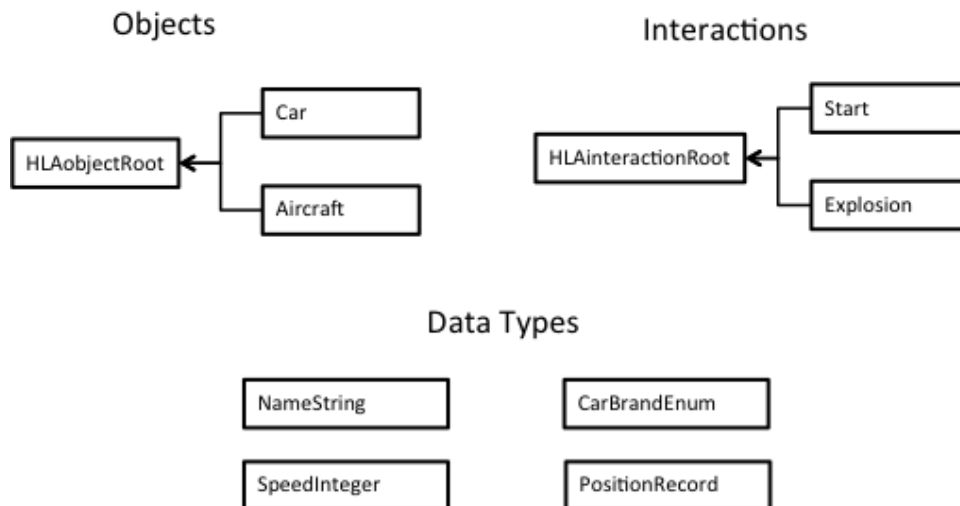


Figure 2-4: Sample contents of an HLA FOM

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.

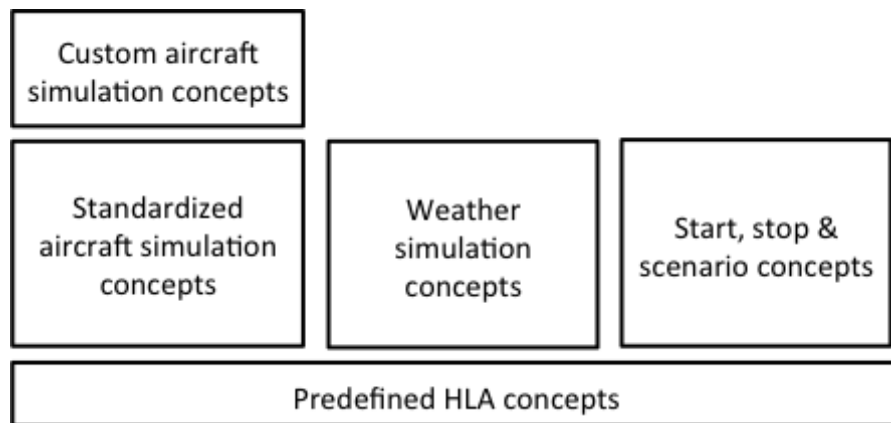


Figure 2-5: FOM Modules

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.

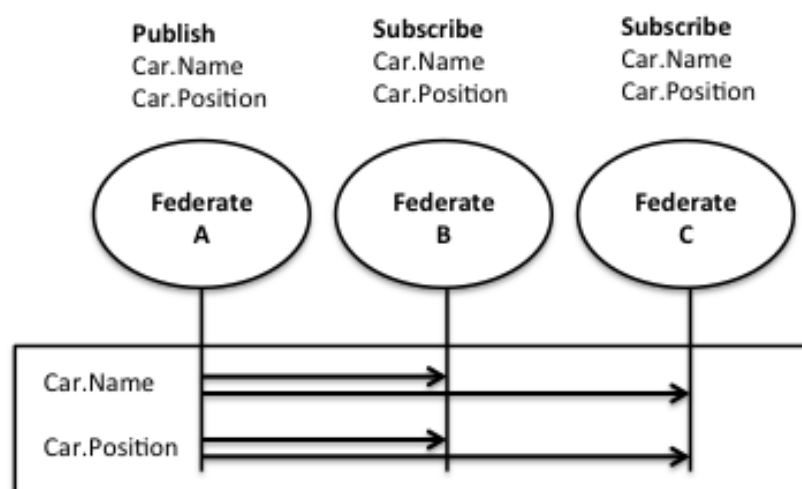


Figure 3-1: Publish and Subscribe

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.

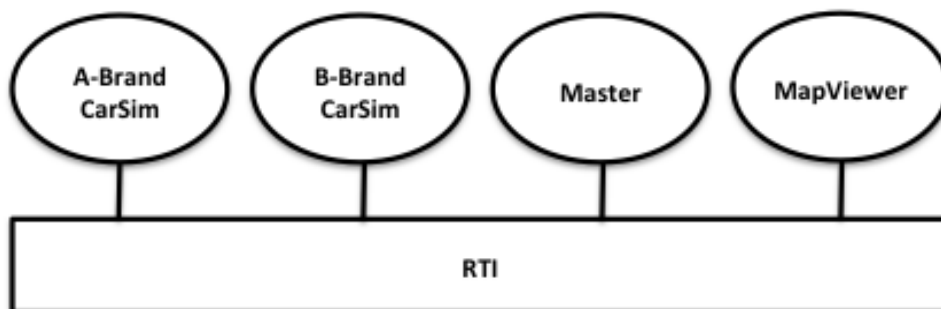


Figure 4-1: The Fuel Economy Federation

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.

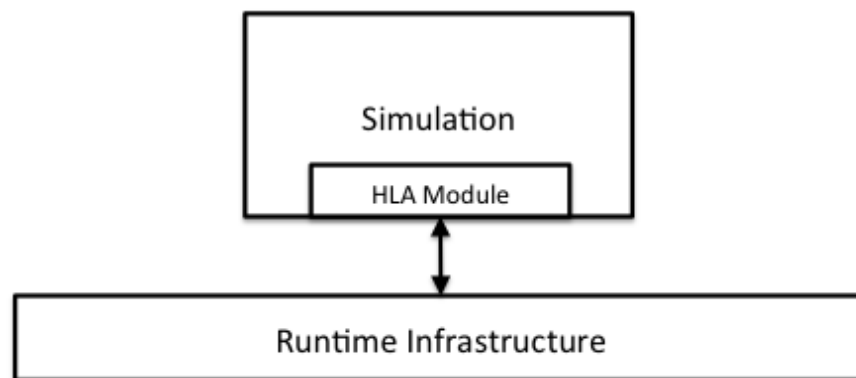


Figure 4-2: The HLA module of a simulation

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:

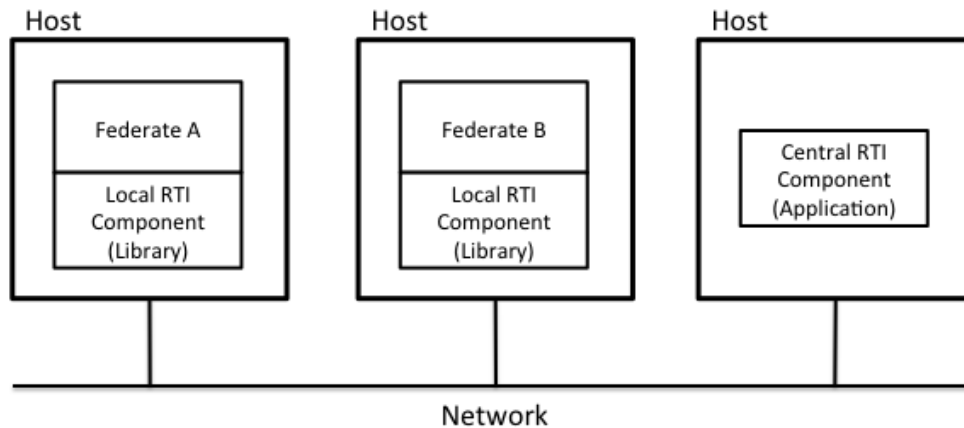


Figure 4-3: The CRC and the LRC

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.

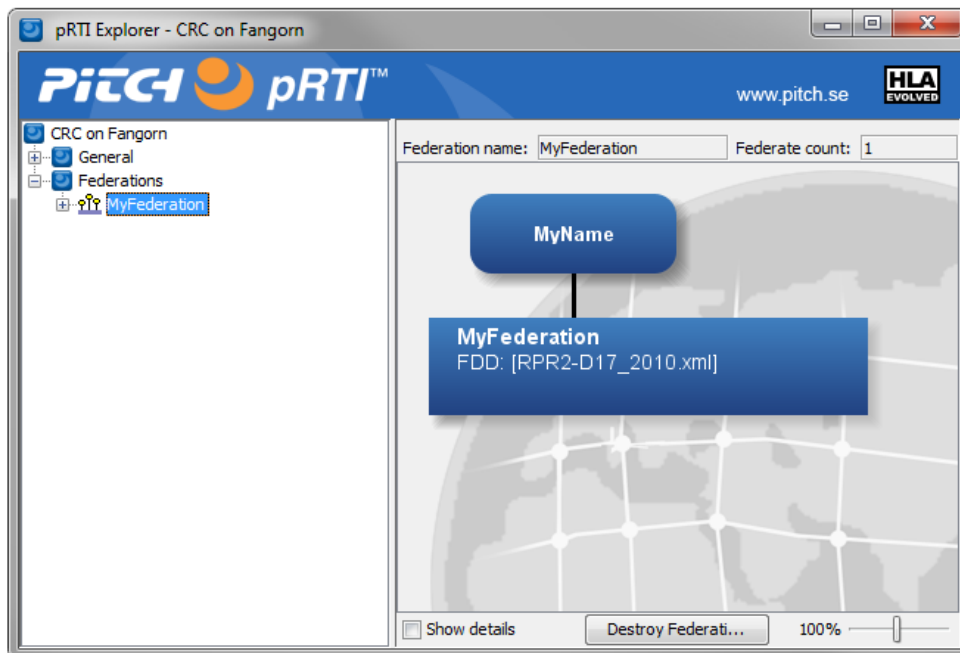


Figure 4-3: The federate and the federation in the RTI GUI

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.

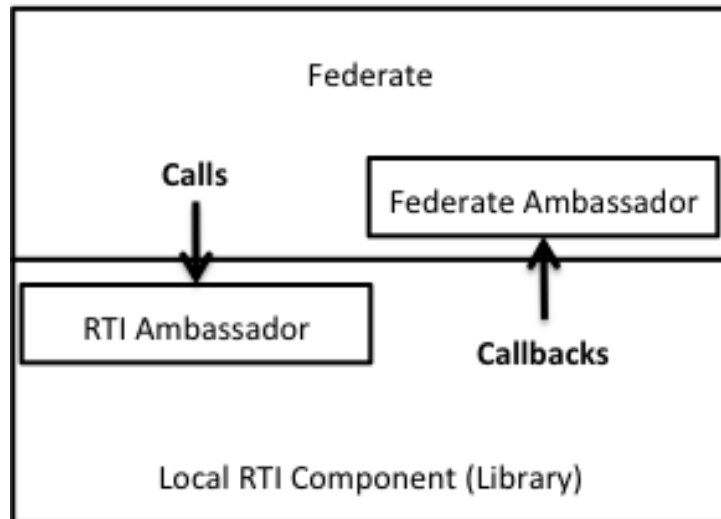


Figure 4-4: Calls, callbacks and ambassadors

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.

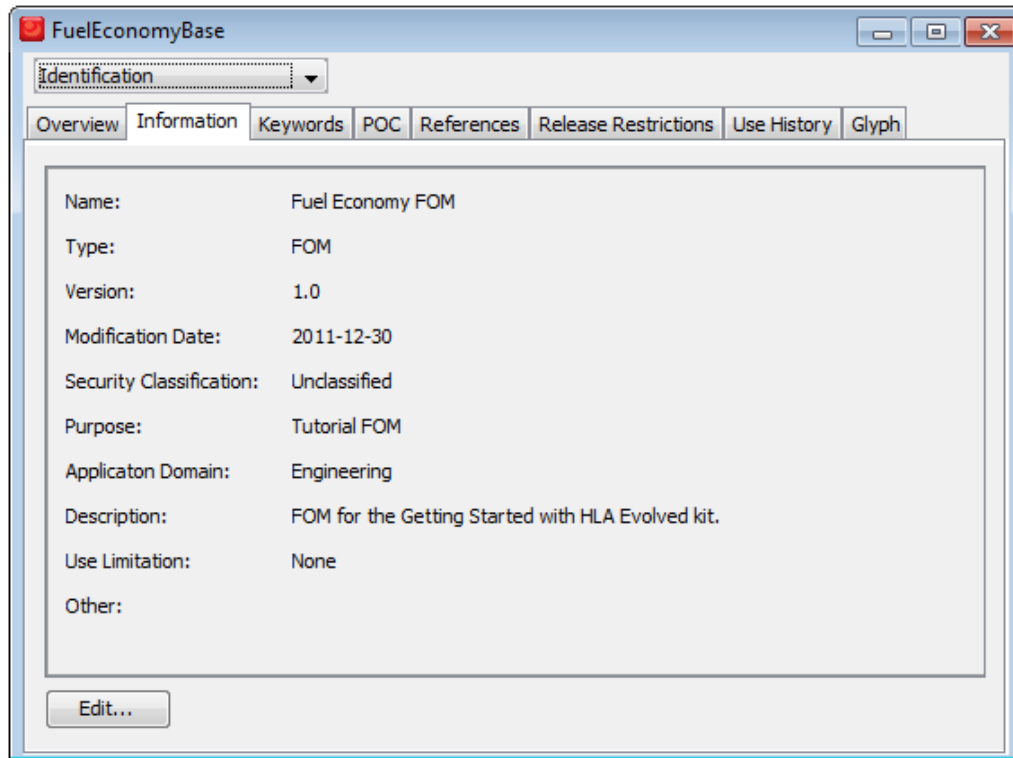


Figure 5-1: Identification table

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.

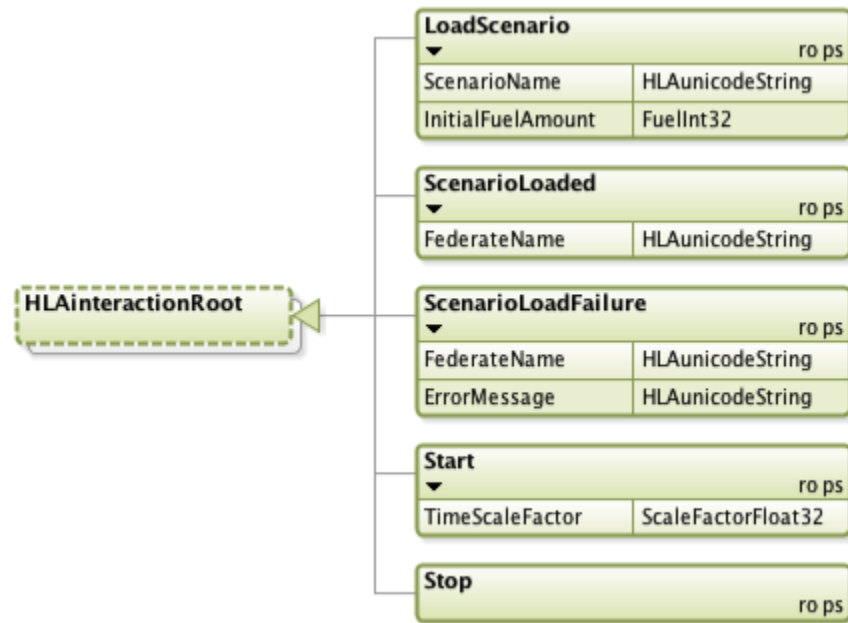


Figure 5-2: Interaction classes

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.

Edit Interaction Class - FuelEconomyBase

Name:

Transportation:

Parameters:

Name	Datatype	Semantics
ScenarioName	HLAUnicodeString	Scenario file for the simula...
InitialFuelAmount	FuelInt32	Initial amount of fuel

☐ Show inherited

Semantics:

Dimensions:

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:

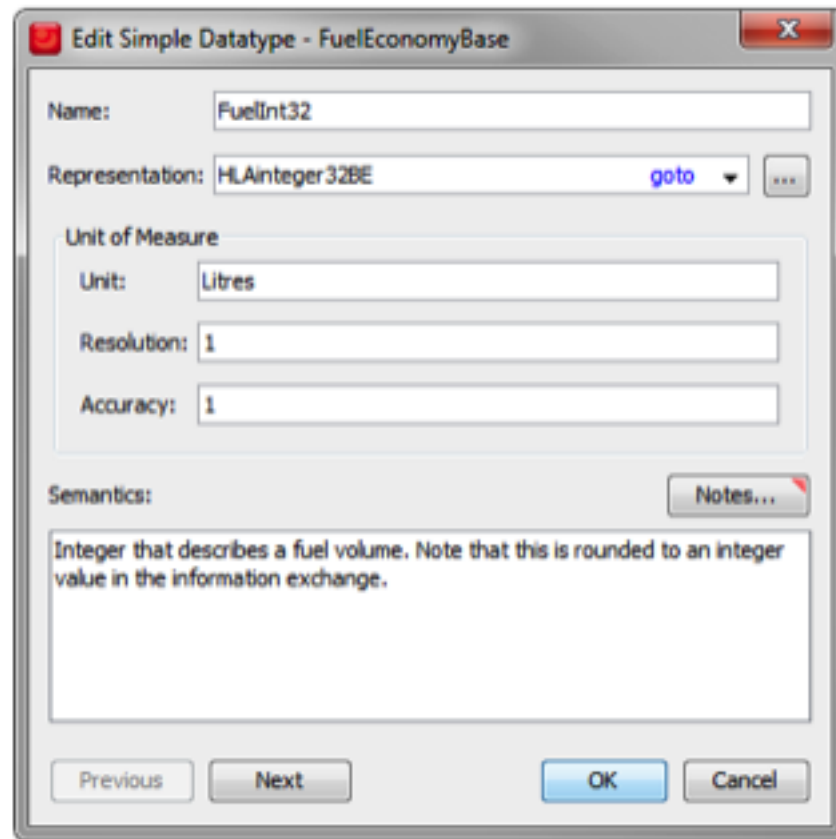


Figure 5-4: The FuelInt32 Simple data type

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.

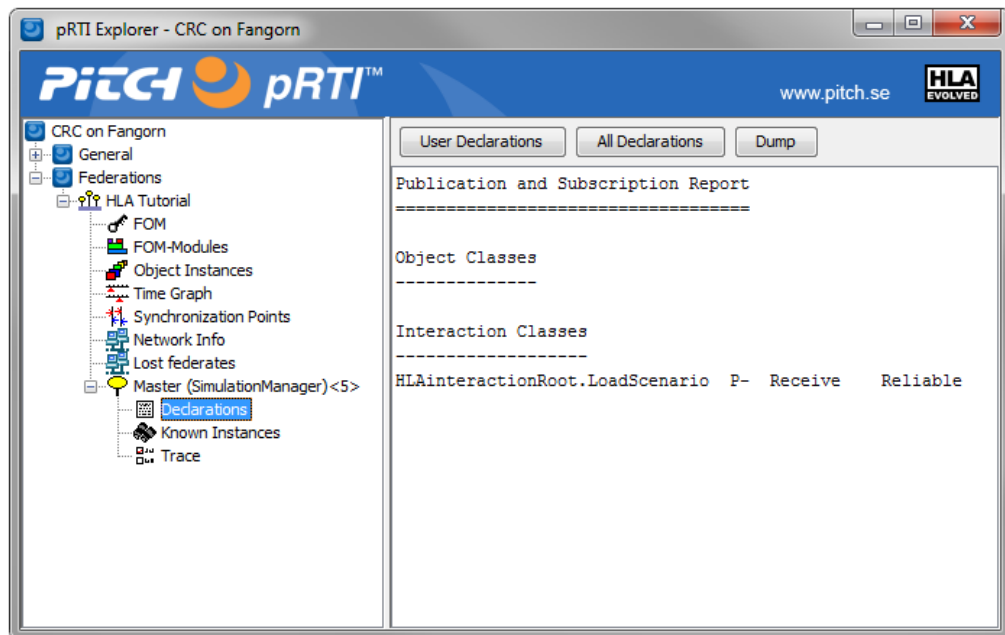


Figure 6-1: An interaction class has been published as seen in the RTI GUI

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:

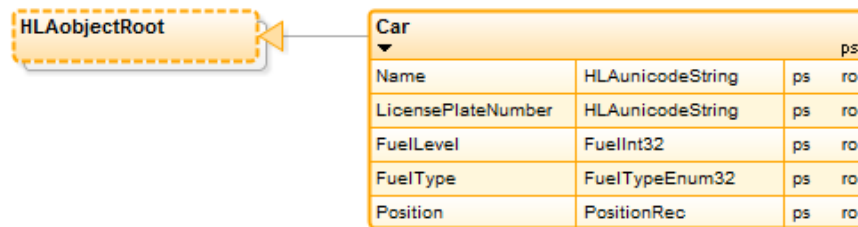


Figure 7-1: Object Classes

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

Edit Object Class - FuelEconomyBase

Name: Car

Sharing: ☒ Publish ☒ Subscribe

Attributes:

Name	Datatype	Publish	Subscribe	Semantics
Name	HLAUnicodeString	☒	☒	Name of the car
LicensePlateNumber	HLAUnicodeString	☒	☒	Number on the license plate.
FuelLevel	FuelInt32	☒	☒	Current fuel level of the car at each point in time
FuelType	FuelTypeEnum32	☒	☒	Type of fuel for the car.
Position	PositionRec	☒	☒	Position of the car.

☐ Show inherited

Semantics:

A car for the Fuel Economy federation

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 closer look at the Name attribute of an object class:

Edit Attribute - FuelEconomyBase

Name:

Class:

Datatype: [goto](#) ...

Sharing: ☒ Publish ☒ Subscribe

Ownership: ☐ Divest ☐ Acquire

Update

Update type:

Update Condition:

Distribution

Order: ☒ Receive ☐ Timestamp

Transportation:

Dimensions:

Semantics: [Notes...](#)

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:

The screenshot shows a dialog box titled "Edit Attribute - FuelEconomyBase". It contains the following fields and controls:

- Name:** FuelType
- Class:** Car
- Datatype:** FuelTypeEnum32 (with a "goto" link and a menu icon)
- Sharing:** ☒ Publish, ☒ Subscribe
- Ownership:** ☐ Divest, ☐ Acquire
- Update:**
 - Update type:** Static (dropdown menu)
 - Update Condition:** NA
- Distribution:**
 - Order:** ☒ Receive, ☐ Timestamp
 - Transportation:** HLAreliable (dropdown menu)
 - Dimensions:** (empty box)
- Semantics:** Type of fuel for the car. (text area)
- Buttons:** Previous, Next, OK, Cancel

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 using the data type FuelTypeEnum32. Let's take a closer look at it:

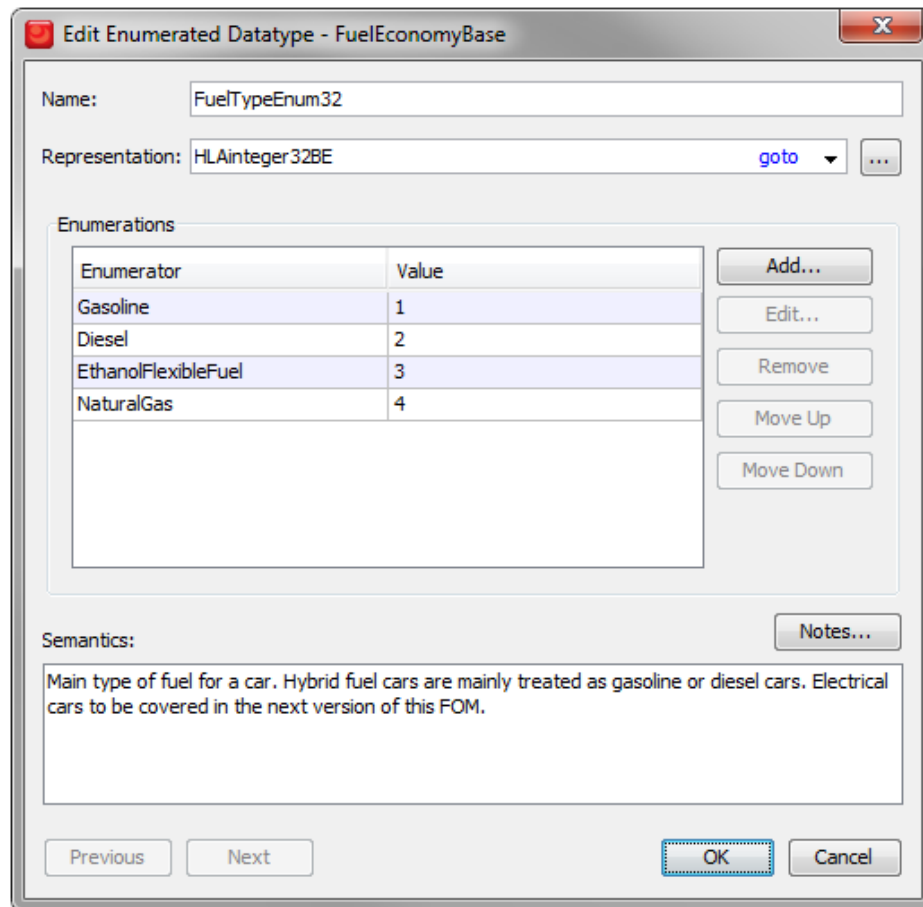


Figure 7-5: The Fuel Type enumerated value

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.

Edit Attribute - FuelEconomyBase

Name:

Class:

Datatype: [goto](#) ...

Sharing: ☒ Publish ☒ Subscribe

Ownership: ☐ Divest ☐ Acquire

Update

Update type:

Update Condition:

Distribution

Order: ☒ Receive ☐ Timestamp

Transportation:

Dimensions:

Semantics: [Notes...](#)

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:

Dialog box titled "Edit Fixed Record Datatype - FuelEconomyBase".

Name:

Encoding:

Fields

Name	Type	Semantics
Lat	AngleFloat64	Latitude degree (-90 to 90)
Long	AngleFloat64	Longitude degree (-180 to 180)

Buttons: Add..., Edit..., Remove, Move Up, Move Down

Semantics:

Buttons: Previous, Next, OK, Cancel

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:

The screenshot shows a Windows-style dialog box titled "Edit Simple Datatype - FuelEconomyBase". It contains the following fields and controls:

- Name:** A text box containing "AngleFloat64".
- Representation:** A dropdown menu showing "HLAfloat64BE" with a "goto" link and an ellipsis button.
- Unit of Measure:** A section containing three text boxes:
 - Unit:** "Degree"
 - Resolution:** "0.000001"
 - Accuracy:** "0.000001"
- Semantics:** A section with a "Notes..." button and a text area containing the text: "Decimal degree data type used for Lat/Long positioning. A 64 bit value is required to accurately describe the position of a car on earth.".
- Buttons:** "Previous", "Next", "OK", and "Cancel" at the bottom.

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.

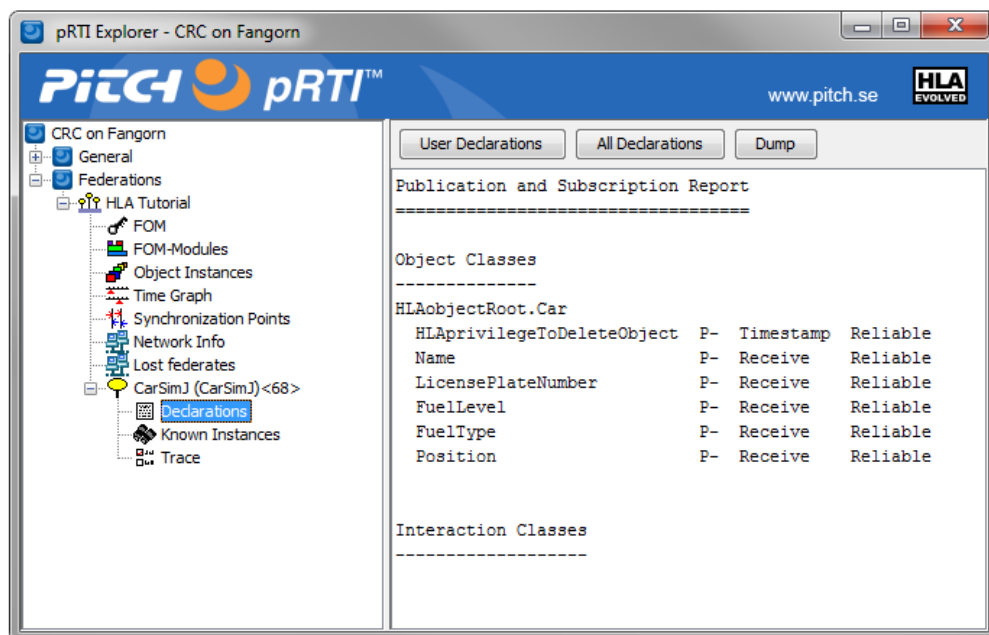


Figure 8-1: Publication of an object class with attributes as seen in the RTI GUI

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.

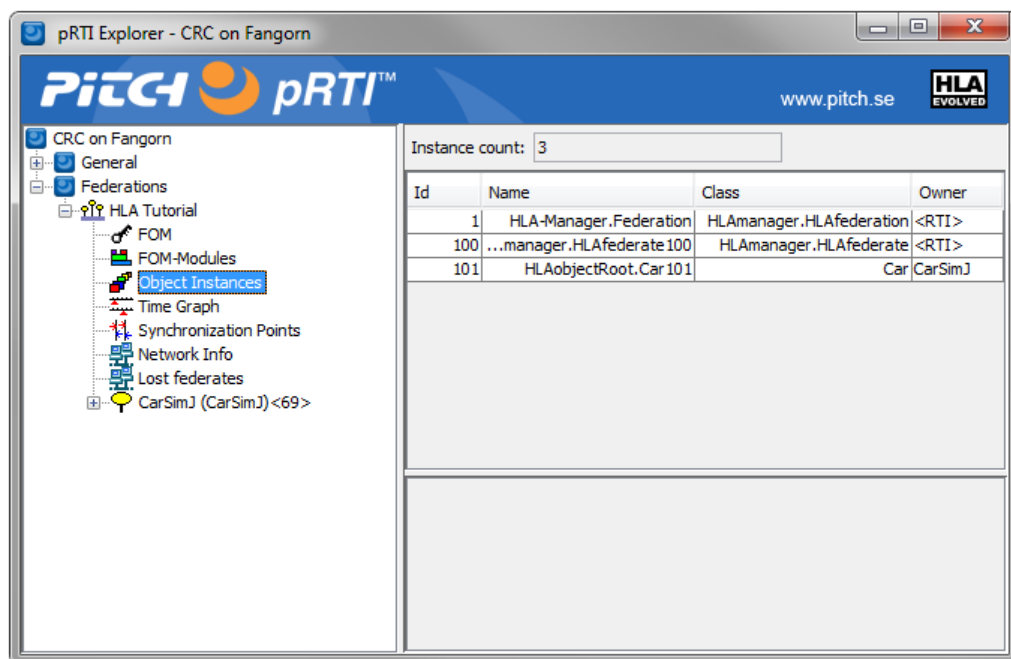


Figure 8-2: A registered object instance as seen in the RTI GUI

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 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 an update.
- Your federate receives the ReflectAttributeValues with updated attribute values for attributes that it 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.

The screenshot shows a dialog box titled "Edit Attribute - FuelEconomyBase". It contains the following fields and controls:

- Name:** Position
- Class:** Car
- Datatype:** PositionRec (with a "goto" link and a menu icon)
- Sharing:** ☒ Publish, ☒ Subscribe
- Ownership:** ☐ Divest, ☐ Acquire
- Update:**
 - Update type:** Conditional
 - Update Condition:** On change
- Distribution:**
 - Order:** ☒ Receive, ☐ Timestamp
 - Transportation:** HLAreliable
 - Dimensions:** (empty box)
- Semantics:** Position of the car. (with a "Notes..." button)

At the bottom are buttons for "Previous", "Next", "OK", and "Cancel".

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 then 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 user's 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
    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
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.

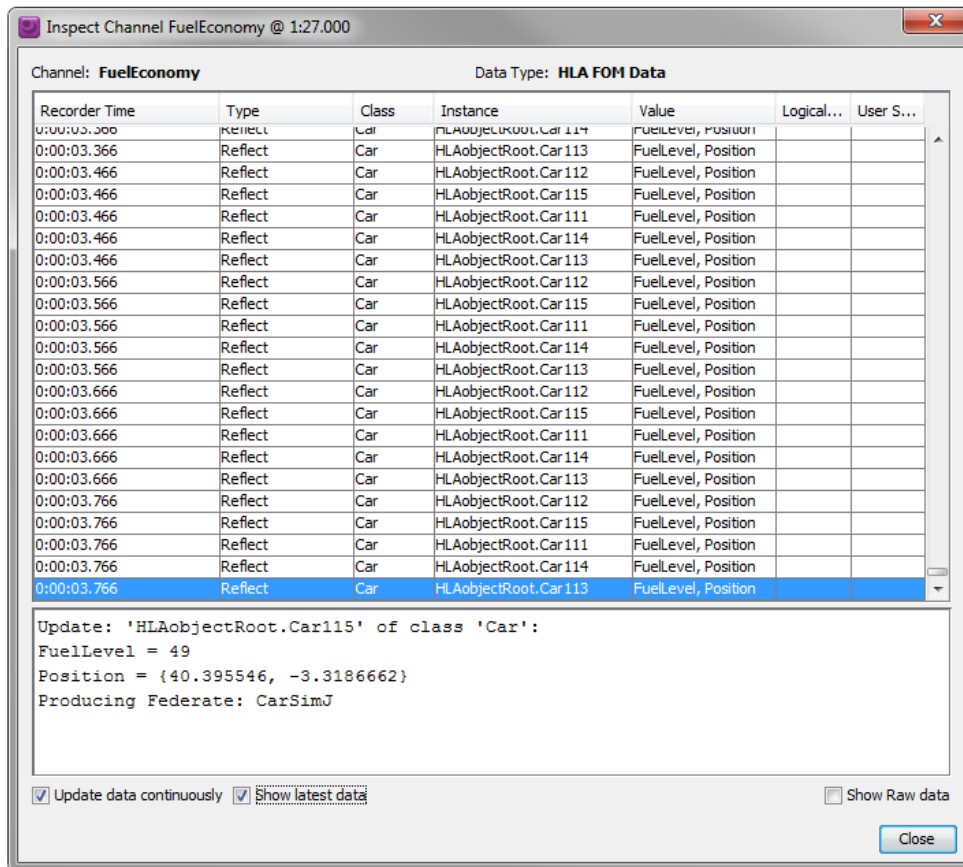


Figure 10-1: Logging data from the Fuel Economy Federation

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.

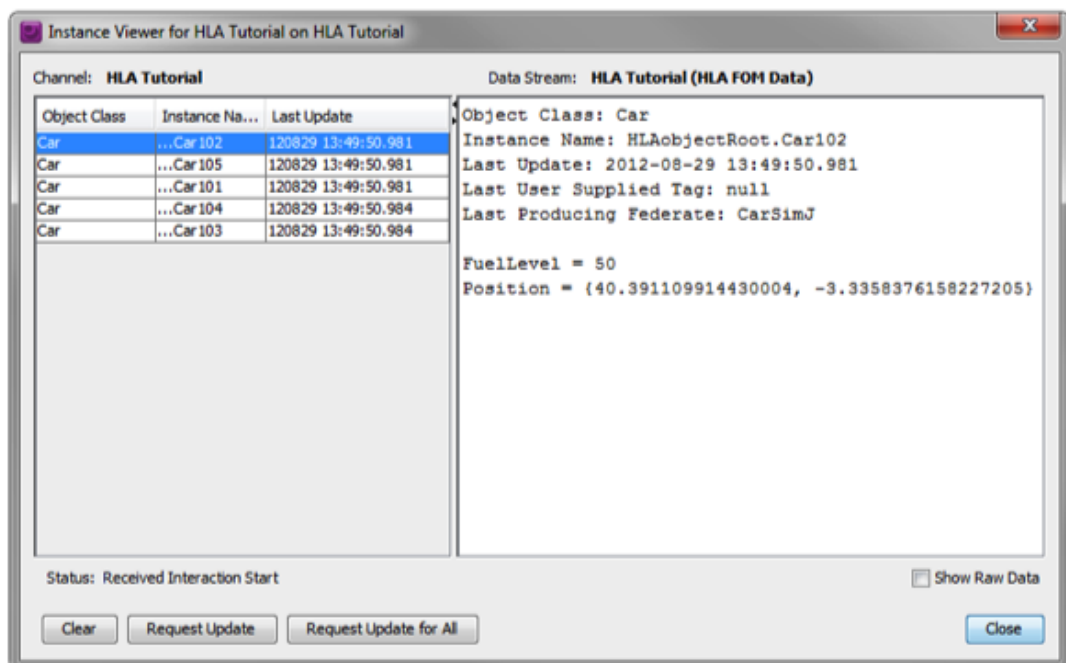
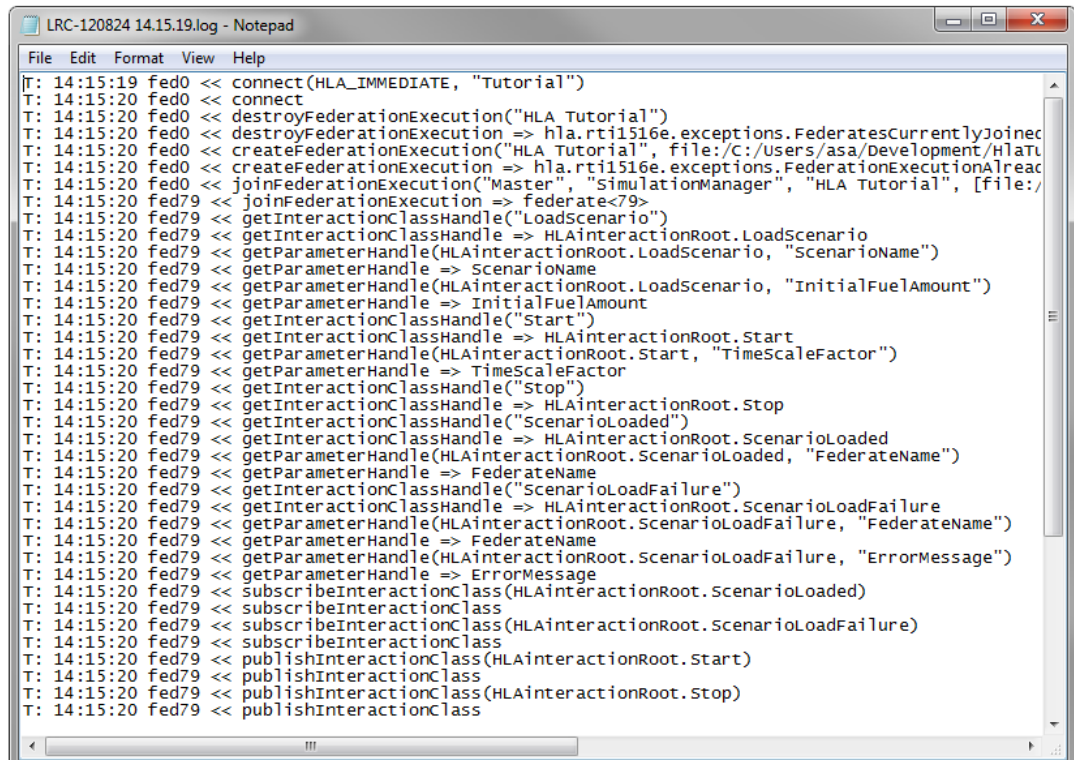


Figure 10-2: Instance viewer of a data logger

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.



```

LRC-120824 14.15.19.log - Notepad
File Edit Format View Help
T: 14:15:19 fed0 << connect(HLA_IMMEDIATE, "Tutorial")
T: 14:15:20 fed0 << connect
T: 14:15:20 fed0 << destroyFederationExecution("HLA Tutorial")
T: 14:15:20 fed0 << createFederationExecution("HLA Tutorial", file:/C:/Users/asa/Development/HLA...
T: 14:15:20 fed0 << createFederationExecution => hla.rti1516e.exceptions.FederationExecutionAlready...
T: 14:15:20 fed0 << joinFederationExecution("Master", "SimulationManager", "HLA Tutorial", [file:/...
T: 14:15:20 fed79 << joinFederationExecution => federate<79>
T: 14:15:20 fed79 << getInteractionClassHandle("LoadScenario")
T: 14:15:20 fed79 << getInteractionClassHandle => HLAinteractionRoot.LoadScenario
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.LoadScenario, "ScenarioName")
T: 14:15:20 fed79 << getParameterHandle => ScenarioName
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.LoadScenario, "InitialFuelAmount")
T: 14:15:20 fed79 << getParameterHandle => InitialFuelAmount
T: 14:15:20 fed79 << getInteractionClassHandle("Start")
T: 14:15:20 fed79 << getInteractionClassHandle => HLAinteractionRoot.Start
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.Start, "TimeScaleFactor")
T: 14:15:20 fed79 << getParameterHandle => TimeScaleFactor
T: 14:15:20 fed79 << getInteractionClassHandle("Stop")
T: 14:15:20 fed79 << getInteractionClassHandle => HLAinteractionRoot.Stop
T: 14:15:20 fed79 << getInteractionClassHandle("ScenarioLoaded")
T: 14:15:20 fed79 << getInteractionClassHandle => HLAinteractionRoot.ScenarioLoaded
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.ScenarioLoaded, "FederateName")
T: 14:15:20 fed79 << getParameterHandle => FederateName
T: 14:15:20 fed79 << getInteractionClassHandle("ScenarioLoadFailure")
T: 14:15:20 fed79 << getInteractionClassHandle => HLAinteractionRoot.ScenarioLoadFailure
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.ScenarioLoadFailure, "FederateName")
T: 14:15:20 fed79 << getParameterHandle => FederateName
T: 14:15:20 fed79 << getParameterHandle(HLAinteractionRoot.ScenarioLoadFailure, "ErrorMessage")
T: 14:15:20 fed79 << getParameterHandle => ErrorMessage
T: 14:15:20 fed79 << subscribeInteractionClass(HLAinteractionRoot.ScenarioLoaded)
T: 14:15:20 fed79 << subscribeInteractionClass
T: 14:15:20 fed79 << subscribeInteractionClass(HLAinteractionRoot.ScenarioLoadFailure)
T: 14:15:20 fed79 << subscribeInteractionClass
T: 14:15:20 fed79 << publishInteractionClass(HLAinteractionRoot.Start)
T: 14:15:20 fed79 << publishInteractionClass
T: 14:15:20 fed79 << publishInteractionClass(HLAinteractionRoot.Stop)
T: 14:15:20 fed79 << publishInteractionClass

```

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:

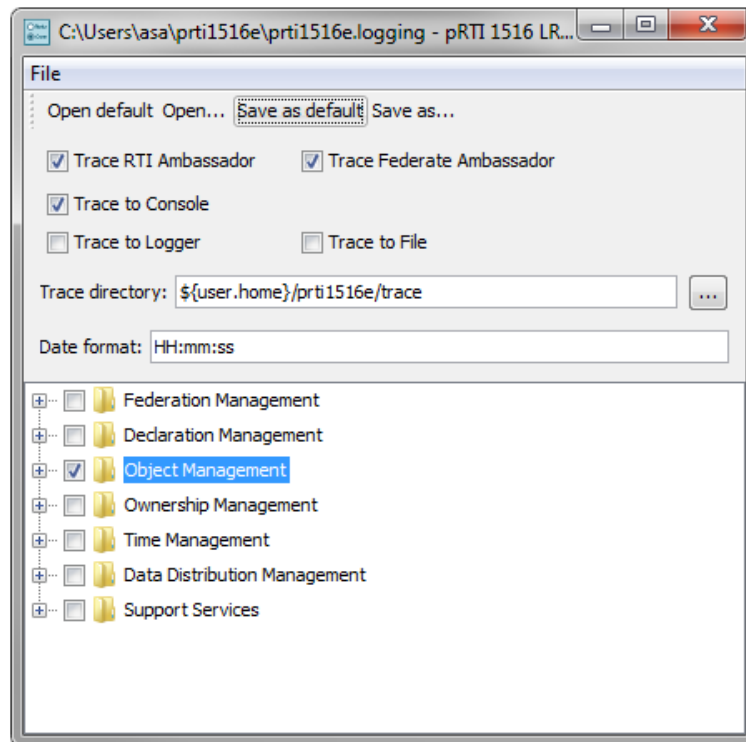


Figure 10-4: Selecting HLA services to trace

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 your 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 it 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 subscribe 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:

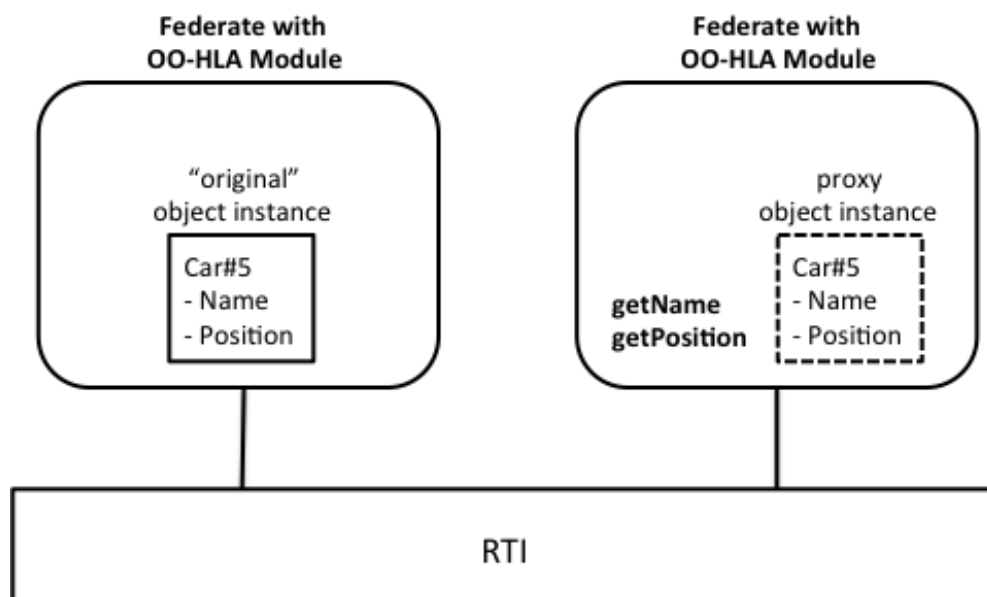


Figure 11-1: Get Value from Proxy object

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.

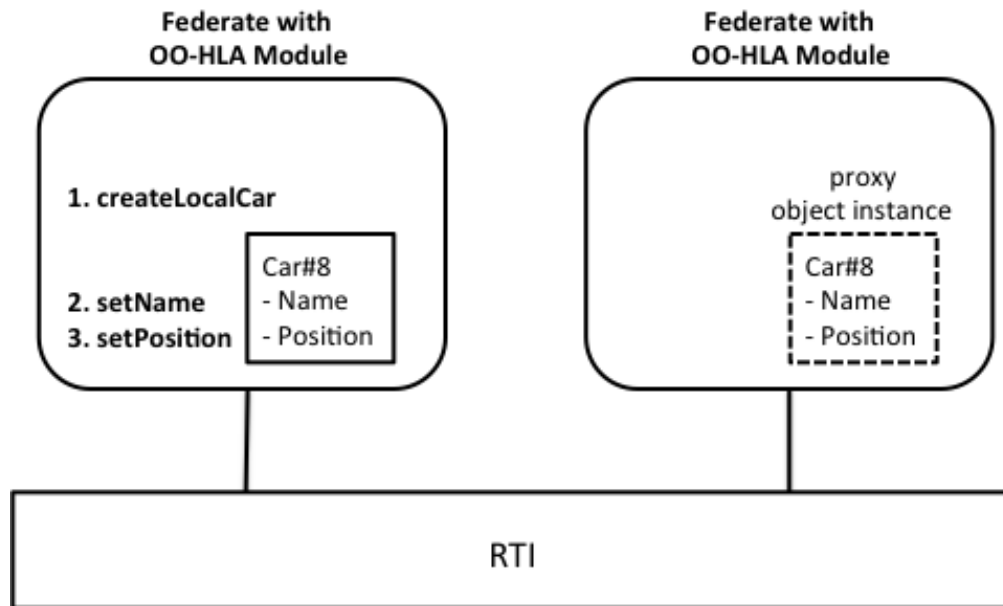


Figure 11-2: Set Value on Local object

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.

Object Class Settings

Give the object class a name to be used in the generated code. Decide if you want the code to support local and/or remote instances. Select the attributes that are of interest and give them name to be used in the generated code. For each attribute you may also specify whether it may be used as a lookup key. You may also specify whether a value needs to be provided when creating a local instance and if a value needs to be available before discovering a remote instance.

Name:

FOM Name: HLAobjectRoot.Car

Support Local and Remote Instances

☐ Only Local

☐ Only Remote

☒ Local and Remote

Attributes:

FOM Name	C++/Java Name	Lookup	Create	Initialized	Always Send
Name	name	☒	☒	☒	☐
LicensePlateNumber	licensePlateNumber	☒	☒	☒	☐
FuelLevel	fuelLevel	☐	☐	☐	☐
FuelType	fuelType	☐	☒	☒	☐
Position	position	☐	☐	☒	☐

Add... Remove...

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.

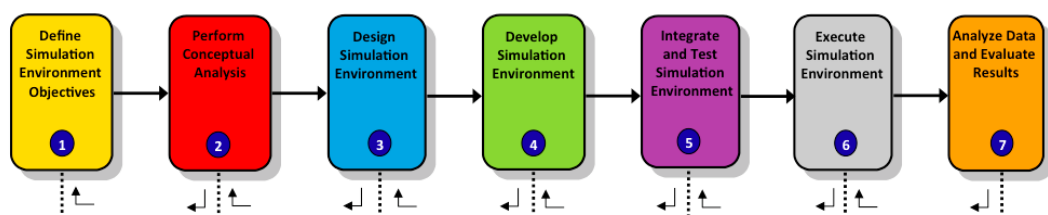


Figure 12-1: DSEEP Process for Simulation Interoperability

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
MapView	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:

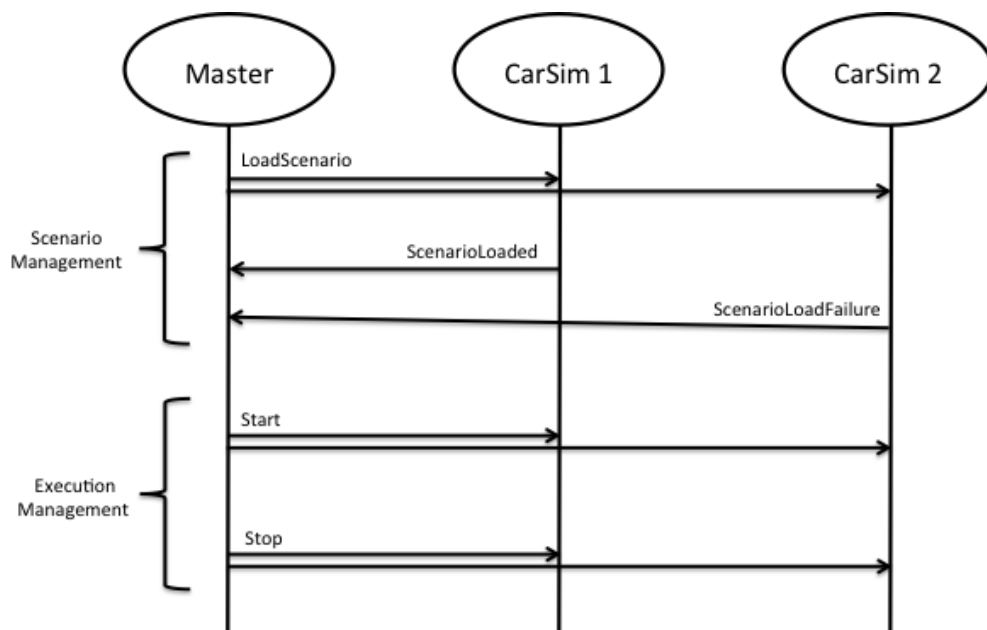


Figure A-1: Scenario and execution management

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 then 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 then 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.

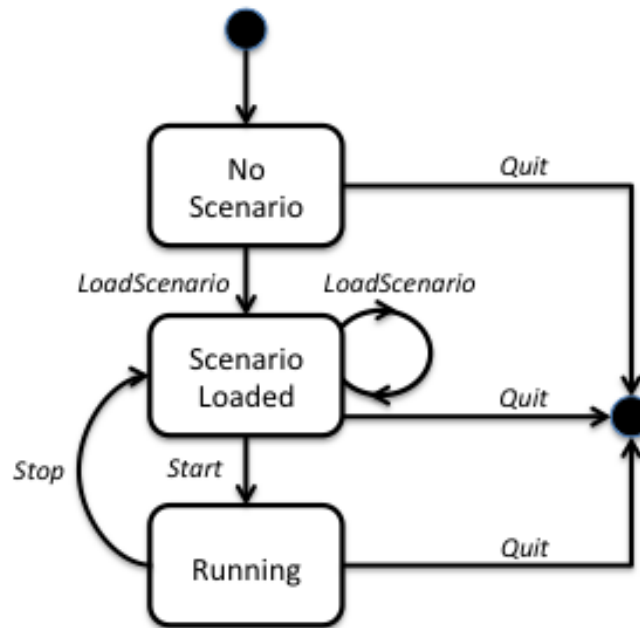


Figure C-1: Internal states in a CarSim federate

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 not 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:

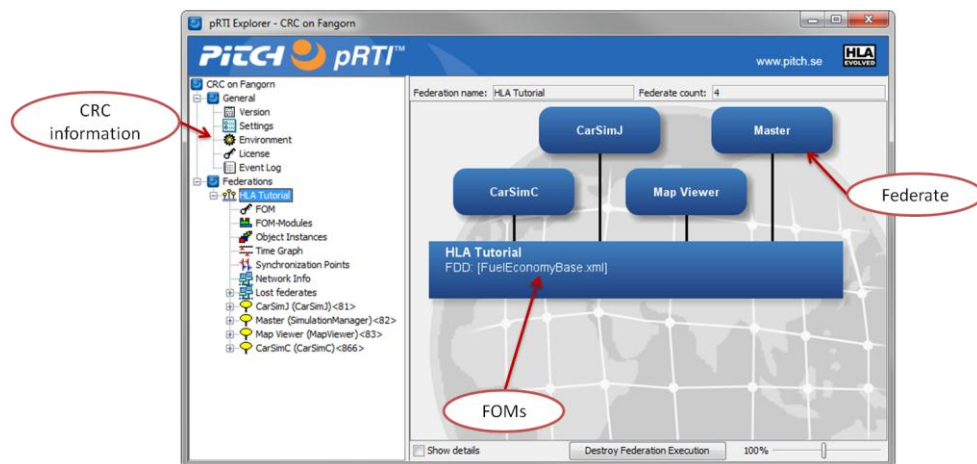


Figure D2-1: The Pitch pRTI user interface

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:

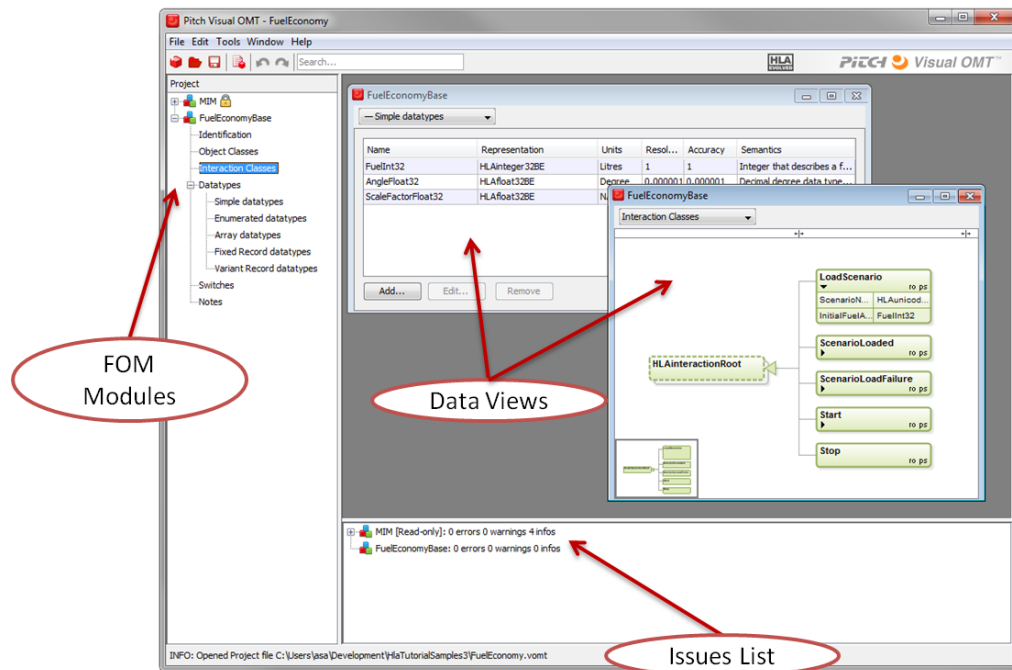


Figure D3-1: The Pitch Visual OMT user interface

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.

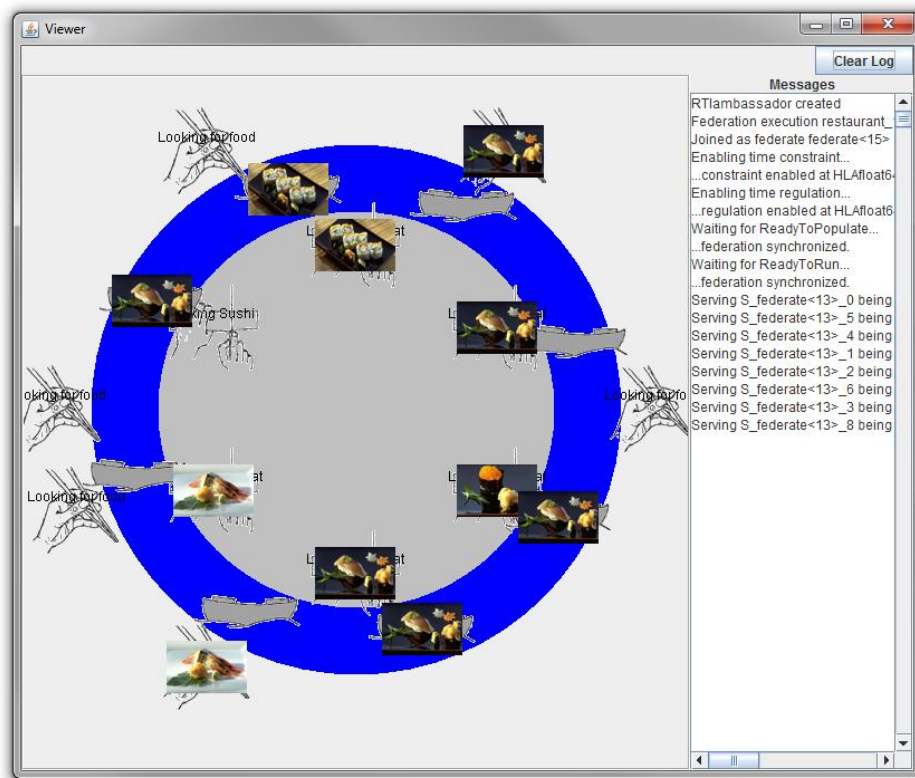
Appendix E: The Sushi Restaurant federation

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 it (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