RETURN BIDS TO:
RETOURNER LES SOUMISSIONS À:

Bid Receiving Public Works and Government Services Canada/Réception des soumissions Travaux publics et Services gouvernementaux Canada
1713 Bedford Row
Halifax, N.S./Halifax, (N.É.)
B3J 1T3
Bid Fax: (902) 496-5016

SOLICITATION AMENDMENT
MODIFICATION DE L'INVITATION

The referenced document is hereby revised; unless otherwise indicated, all other terms and conditions of the Solicitation remain the same.

Ce document est par la présente révisé; sauf indication contraire, les modalités de l'invitation demeurent les mêmes.

Comments - Commentaires

Title - Sujet PROGRAMMING WORK	
Solicitation No. - N° de l'invitation W7707-135657/A	**Amendment No. - N° modif.** 002
Client Reference No. - N° de référence du client W7707-13-5657	**Date** 2013-02-21
GETS Reference No. - N° de référence de SEAG PW-$HAL-220-8906	
File No. - N° de dossier HAL-2-69336 (220)	**CCC No./N° CCC - FMS No./N° VME**
Solicitation Closes - L'invitation prend fin **at - à 02:00 PM** **on - le 2013-02-26**	**Time Zone** **Fuseau horaire** Atlantic Standard Time AST
F.O.B. - F.A.B. **Plant-Usine:** ☑ **Destination:** ☐ **Other-Autre:** ☐	
Address Enquiries to: - Adresser toutes questions à: Dunphy, Nancy	**Buyer Id - Id de l'acheteur** hal220
Telephone No. - N° de téléphone (902) 496-5481 ( )	**FAX No. - N° de FAX** (902) 496-5016
Destination - of Goods, Services, and Construction: **Destination - des biens, services et construction:**	

Vendor/Firm Name and Address
Raison sociale et adresse du fournisseur/de l'entrepreneur

Instructions: See Herein

Instructions: Voir aux présentes

Delivery Required - Livraison exigée	**Delivery Offered - Livraison proposée**

Vendor/Firm Name and Address
Raison sociale et adresse du fournisseur/de l'entrepreneur

Telephone No. - N° de téléphone
Facsimile No. - N° de télécopieur

Name and title of person authorized to sign on behalf of Vendor/Firm (type or print)
Nom et titre de la personne autorisée à signer au nom du fournisseur/ de l'entrepreneur (taper ou écrire en caractères d'imprimerie)

Signature **Date**

Issuing Office - Bureau de distribution

Acquisitions
1713 Bedford Row
Halifax, N.S./Halifax, (N.É.)
B3J 3C9

Canada

Cet amendement 002 est soulevée à ajouter la version préliminaire du

Voir ci-joint.

Ce document provient d'une tierce partie et n'est pas availble si le français..

A Toolkit for Building RTI Independent HLA Interfaces for Simulations

Allan D. Gillis
Senior Scientific Programmer / Analyst
Virtual Combat Systems Group
Maritime Information and Combat Systems
Defence R&D Canada - Atlantic

Glenn P. Franck
Scientific Programmer / Analyst
Virtual Combat Systems Group
Maritime Information and Combat Systems
Defence R&D Canada - Atlantic

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM [enter number only: 9999-999]
March 2009

Principal Author

Original signed by Allan D. Gillis

Allan D. Gillis

Senior Scientific Programmer / Analyst

Approved by

Original signed by [Approved By Name]

[Approved By Name]

[Approved By Position/Title]

Approved for release by

Original signed by [Released By Name]

[Released By Name]

[Released By Position/Title]

[Include the sponsor of the work or a reference to a thrust or work unit, when deemed appropriate by author or CSA; relevant patent number(s), relating to protected intellectual property, should be noted. If there is no relevant information for this document, delete this text.]

Abstract

[Enter text: English]

Résumé

[Enter text: French]

This page intentionally left blank.

Executive summary

A Toolkit for Building RTI Independant HLA Interfaces for Simulations:

Allan D. Gillis, Glenn P. Franck; DRDC Atlantic TM [enter number only: 9999-999]; Defence R&D Canada – Atlantic; March 2009.

Introduction or background: [Enter text: English]

Results: [Enter text: English]

Significance: [Enter text: English]

Future plans: [Enter text: English]

Sommaire

A Toolkit for Building RTI Independant HLA Interfaces for Simulations:

Allan D. Gillis, Glenn P. Franck~~Allan D. Gillis~~; DRDC Atlantic TM [enter number only: 9999-999]; R & D pour la défense Canada – Atlantique; Mars 2009.

Introduction ou contexte: [Enter text: French]

Résultats: [Enter text: French]

Importance: [Enter text: French]

Perspectives: [Enter text: French]

This page intentionally left blank.

Table of contents

Abstract ……. i
Résumé ….... i
Executive summary iii
Sommaire iv
Table of contents vi
List of figures viii
List of tables ix
Acknowledgements x
1 Introduction..... 1
2 Overall Design Concept..... 3
2.1 Maintenance 4
2.1.1 RTI Vendor and HLA Abstraction Layers..... 4
2.1.2 Federation Layer 5
2.1.3 Simulation Layer..... 5
2.2 Multi-thread / process safety 5
3 HLA Abstraction Layer 6
3.1 The Base API..... 6
3.2 Converting and Adapting 6
4 Federation Encapsulation Layer 9
4.1 Structure 9
4.2 Module duties..... 9
4.3 Layer API 9
4.4 Module developer guidance..... 10
5 Simulation Layer 11
6 FOM Library Requirements..... 12
6.1 Encoding and Decoding Support..... 12
6.2 FOM Object and Interaction Classes..... 12
6.3 Developer Productivity..... 12
6.4 Strong Typing..... 13
6.5 Allowing Developer Extensions..... 13
6.6 Code Generation..... 14
7 Library Design..... 15
7.1 Data types 15
7.2 FOM objects and interactions..... 16
7.2.1 HLA object classes..... 16

7.2.2 HLA interaction classes 18
7.2.3 Design of the library..... 18
7.2.4 What about FOM interactions? 23
7.3 Java packages 23
References 26
Annex A[Enter Annex Level 1 heading here] 28
A.1[Enter Annex Level 2 heading here] 28
A.1.1[Enter Annex Level 3 heading here]..... 28
Bibliography 30
List of symbols/abbreviations/acronyms/initialisms 31
Glossary 32
Index …… 33
Distribution list 34

List of figures

Figure 1, overall architecture. 3
Figure 2, how the HLA Abstraction Layer hides vendor implementations. 6
Figure 3, adapting vendor libraries to the HLA Abstraction Layer. 7
Figure 4, VendorAdaptor interface. 7
Figure 5, sample federation modules in the encapsulation layer. 9
Figure 6, object class structure in HLA FOMs. 16
Figure 7, example hierarchy from the RPR 2.0 Draft 18 FOM. 17
Figure 8, example interaction hierarchy from the RPR 2.0 Draft 18 FOM. 18
Figure 9, one possible method that combines interfaces with inheritance. 19
Figure 10, FOM Object class diagram showing the behaviour interface. 20
Figure 11, an example showing the Behaviour and FOMClass inheritance for a RPR 2 based FOM library. 22
Figure 12, generic package structure of a FOM library. 24
Figure 13, example package diagram for the RPR2 Draft 18 FOM showing all packages. 25
Figure A-14: This is the caption for the figure shown above. 28

List of tables

No table of figures entries found.

Acknowledgements

[Enter Acknowledgements here if applicable. If not, delete the page.]

This page intentionally left blank.

1 Introduction

The High Level Architecture (HLA) for connecting distributed simulations has been in use since XXX and has evolved significantly over the years to better support the needs of the simulation community.

While HLA concepts are relatively easy to understand, there are significant challenges for any developer working on an HLA federate. First one must understand the fundamental concepts of the architecture, next the intricacies of the Application Programming Interface (API) defined in the various standards must be mastered. It is only after a developer has a firm grasp of the basics of HLA programming that the real problems arise.

The first hurdle is that there are two commonly used versions of HLA, and a third on the way. The only standards-based version of HLA is IEEE 1516 with its various amendments, including the Simulation Interoperability Standards Organisation (SISO) Dynamic Link Compatible (DLC) API standard [3]. Previous to IEEE 1516 there were many variations of HLA all derived from the DMSO versions and collectively referred to as the "1.3" version of HLA. Unfortunately while these various RTI's are referred to collectively they are by no means interchangeable as the API varies from vendor-to-vendor. A federate developer often needs to maintain versions of his federate that are compatible with multiple RTIs.

The second big challenge for a federate developer is the glue that binds all of the simulations together in an HLA distributed simulation (called a federation) together; the Federation Object Model (FOM). The FOM defines the data that can be shared amongst the individual simulations (or "federates" in HLA parlance). This includes basic data representations (16, 32, and 64 bit integers for example) as well as data structures, and the objects which are defined in terms of this data.

The developer, therefore, in addition to writing a simulation and understanding multiple versions of the HLA, must also handle all of the objects, interaction, attributes, parameters, and data types that are defined in the FOM. This usually means writing classes to handle the HLA aspects, and the internal simulation aspects of everything in the FOM that is subscribed to, or published, by the federate being developed.

If two federates use the same FOM and RTI then a lot of code would be duplicated between them. Also, it has been our experience in the MC2CD group and also of others[1] that encoding and decoding the values of attributes/parameters is both tedious and error-prone.

Even though there is much commonality of code there is still a lot that must be different between federates. For example, one simulation may require that it's ships have a "goToFlankSpeed()" method, while another may need to add a "SniperDetectionSystem" field to an armoured vehicle class. It is also quite possible that the problem domain of the simulation will lead to an object structure that has little in common with the FOM (or multiple FOMs) that will be used to participate in HLA federations.

Clearly there is a need for tools to help the developer meet the twin challenges of multiple RTI versions and FOMs while allowing her to work within the problem domain of the system being simulated.

This document describes the design of a set of libraries and tools that will help software developers meet these challenges.

2 Overall Design Concept

The general solution presented in this paper is a framework supported by tools to accomplish three main goals for developers:

1. Eliminate the need to deal with RTI versions and vendor specific quirks at the lowest level.

2. Encapsulate the handling of all FOM and federation agreement specific processing for reuse.

3. Separate the simulation itself from the intricacies of HLA and the specifics of a FOM.

We see the transition from vendor specific RTI libraries to the simulation as a multi-layered architecture, as shown in Figure 1 below.

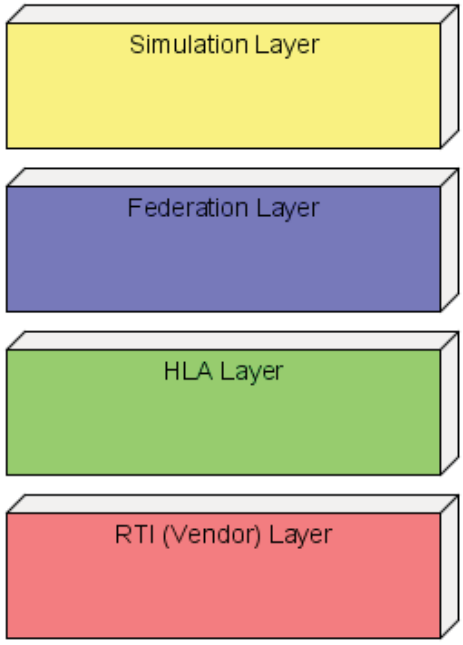

Figure 1, overall architecture.

At the very bottom we have the RTI layer. This layer consists of all the code required to interface with a particular vendor's RTI. The RTI itself may be IEEE 1516 compatible, DMSO 1,3 based, or be compatible with the upcoming HLA Evolved standard when it is available. The problem at this level is that there are differences between the APIs of the various HLA standards (1516 vs

DMSO 1.3 vs MaK 1.3) as well as quirks between vendor implementations of the IEEE 1516 standard (or even between versions from a single vendor).

To alleviate these problems our design inserts an HLA layer between the vendor code and any other part of the system. The HLA layer has a consistent API, so developers working on components for the next layer up don't have to worry about which vendor's RTI will be used.

One layer up is the Federation Layer. This layer contains the compnents needed to actually join a federation using a particular FOM and federation agreement (all the details about how things shall be done that isn't in the FOM). Potentially there will be a separate federation layer component for every FOM / federation agreement combination. For example, the Real Time Platform Reference FOM version 1 would have it's own set of Federation Layer Components, as would the Real Time Platform Reference FOM 2.0.

On top of the Federation Layer is the Simulation Layer. This piece is responsible for adapting any particular simulation to the federation layer underneath. This layer acts as a façade to the more complicated systems below, and converts FOM data to simulation domain data, and vice versa.

2.1 Maintenance

Before moving on to discuss the details of how all these layers will work together it is important to consider the maintenance implications of this approach. In particular, when will a given layer need to be touched by a developer, and what will the consequences be for other layers.

2.1.1 RTI Vendor and HLA Abstraction Layers

Of all the layers the RTI vendor layer is the most likely to change, and is one we have no control over. Vendors will do what they do regardless of what we would like. Changes in the RTI layer may or may not mean changes to the HLA layer, and should have no impact above that (with one notable exception).

First it is worth mentioning that while the RTI layer includes every conceivable RTI version from every possible vendor, there is no obligation upon us to support any given product. New RTI versions or whole new products that we will not use will have no impact on maintainability at all; we shall simply not support them.

A second possible case is a new RTI version that eliminates the quirks of a previous version, or introduces new ones. In this case (always assuming we wish to support the particular version) the HLA layer will require work. For these cases the HLA layer components will be updated by a minor version number (for example 1.0 to 1.1) and the supporting documentation updated. For these types of changes there should be no need to alter layers above.

The third possibility is that an entirely new version of HLA is introduced, or we decide to support something besides HLA (say DIS or TENA). If the new addition to the Vendor Layer significantly alters the core functionality it will become necessary to alter the basic API of the

HLA layer. If (when) this happens the HLA Layer will be updated by a major version (for example 1.2 to 2.0) and compatibility with the federation layer may be lost.

2.1.2 Federation Layer

This layer is not a single entity the way the HLA Abstraction layer is. It is better to think of it as a collection of FOM/Federation Agreement specific modules that provide services to a developer writing a Simulation Façade. Since each module fully encapsulates a particular FOM/Federation Agreement a separate module must be written for each of these.

In theory this may lead to an explosion of Federation Layer modules, but in practice they are likely to be limited in number. In the MC2CD group we really only use VMSA and RPR2 based FOMs meaning we may only need two basic modules that can be derived from for variations required for a specific experiment.

Beyond the need for changes due to alterations in a FOM or Federation agreement, this layer will not need to change. The simulation layer will communicate with this layer through a well defined mechanism that will not alter significantly between versions, and only major changes in functionality in the HLA Layer will break compatibility.

2.1.3 Simulation Layer

Finally there is the Simulation Layer at the top. At this level any changes required will be due to needs of the simulation itself and it's usage. For example, if every single layer below has undergone an update that breaks compatibility it will have no effect on the Simulation Layer unless the particular simulation is required to use that new functionality.

While the simulation layer will have to be altered if additional FOM support is required, this is once again driven by the needs of the simulation developer and not the framework.

2.2 Multi-thread / process safety

One of the most common needs in HLA programming is to bridge two federations using different FOMs to one another, or to join to a simulation that, while "HLA compatible", may not be fully compatible with a larger HLA federation. In these cases, as well as many others, it may be necessary to run multiple instances of the HLA Abstraction Layer within a single application (separate threads), or communicating between separate applications (separate processes).

In order to make the framework inherently multi-thread safe, while still keeping the programming overhead simple, all communication between layers shall occur through a messaging mechanism. In the Java implementation this message system shall utilise the LinkedBlockingQueue [7] found in java.util.concurrent. C++ implementations shall utilise a cross platform open source library that includes a similar capability (ACE [8] for example).

3 HLA Abstraction Layer

As outlined in section2.1.1, this layer of the framework is responsible for removing HLA and RTI version dependency from your federates. The intention is that developers should be able to programme against the HLA Abstraction Layer API and then use any version of any vendor's RTI, and with some limitations, any version of HLA. The basic idea is show in Figure 2.

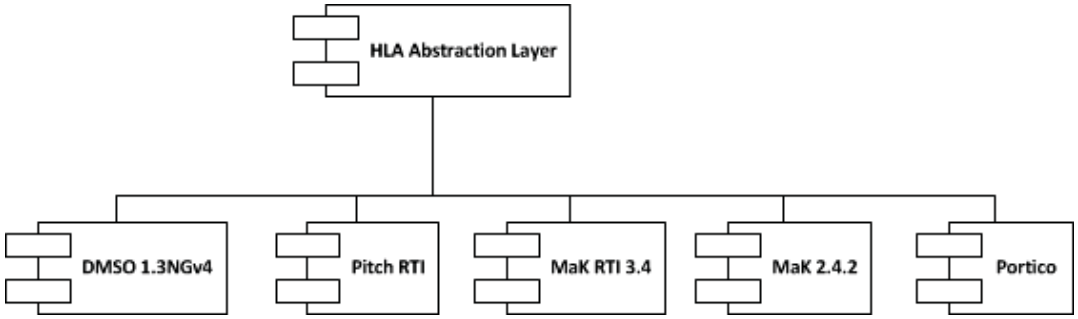

Figure 2, how the HLA Abstraction Layer hides vendor implementations.

In Figure 2, 5 different vendor libraries are show, all hidden by the HLA Abstraction Layer. These libraries are a mixture of HLA versions and vendor implementations, each with their own quirks. The HLA Abstraction Layer removes the need for a developer to deal with the quirks of vendor libraries and the differences in HLA standards.

3.1 The Base API

Our intention is for the HLA Abstraction Layer to appear to be a Dynamic Link Compatible [3], IEEE 1516 standard [5] HLA library. This means the HLA Abstraction layer must implement the entire IEEE 1516 standard correctly, and account for quirks in the vendor implementations as well as be able to convert calls to IEEE 1516 to the various versions of HLA 1.3, or any other standard we decided to support.

Since the API is well described in the IEEE standards, that part of the design will not be discussed any farther here. The developers of this layer shall adhere to the IEEE standards, as well as the SISO DLC standard, including all of the normative and descriptive comments found in those documents.

At run time it will be possible to switch which HLA API is used so that a federate using the HLA Abstraction Layer can connect to any supported vendor's RTI.

3.2 Converting and Adapting

To work with all those differing vendor implementations and standards, every vendor library will require an adaptor. This is shown in Figure 3.

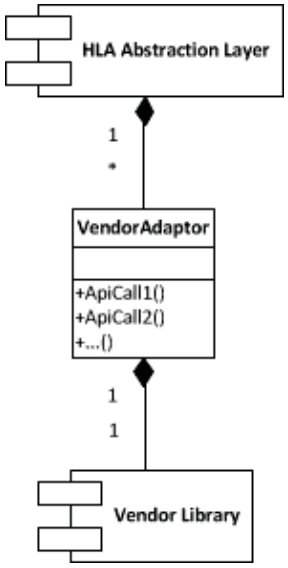

Figure 3, adapting vendor libraries to the HLA Abstraction Layer.

All of these adaptors shall conform to a common interface, as shown in Figure 4.

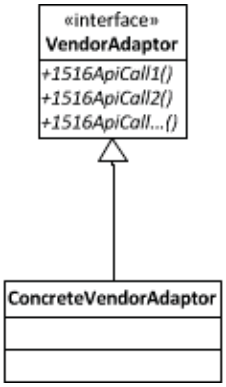

Figure 4, VendorAdaptor interface.

Since the HLA Abstraction Layer is essentially IEEE 1516 + SISO DLC, when using a 1516 RTI, most calls should be simply passed-through to the underlying RTI. When using a non-1516 RTI, say DMSO 1.3NGv4, the vendor adaptor will be responsible for converting the 1516 call into the correct call for the DMSO 1.3NGv4 API.

This will not necessarily be simple, and it is quite possible that some operations available in 1516 will have no equivalent in a different API. In these cases the vendor adaptor should throw exceptions that are fine-grained enough to allow the HLA Abstraction Layer code to deal with problems.

How the HLA version shall be specified has not yet been determined. It could be set via an API call, or included as part of a constructor call for a concrete adaptor, or both. There are advantages to both, and this detail need to be worked out before implementation begins.

4 Federation Encapsulation Layer

As mentioned in section 2 this layer encapsulates a federation, which is a combination of a FOM and a federation agreement. The Simulation Façade makes use of this layer, sending information to it, and receiving information from it.

4.1 Structure

This layer is not monolithic, instead it is composed of federation specific modules that are independent of each other (except via inheritance where appropriate). A Simulation Façade developer chooses which modules to use to provide federation compliance for her simulation; see Figure 5.

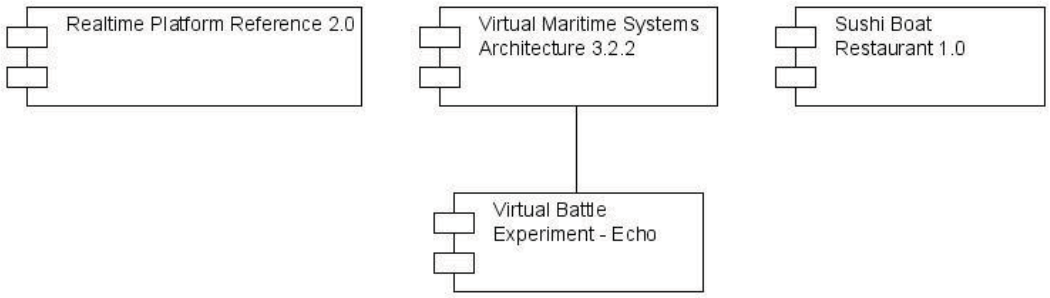

Figure 5, sample federation modules in the encapsulation layer.

4.2 Module duties

The primary duty of any module in this layer is to act as an all-purpose federate for the particular FOM and federation agreement combination it supports. These modules should (ideally) fully support any given FOM with it's

4.3 Layer API

Module developers require an API to make compatible modules. The API is provided as one tool in the total system. While the API includes base classes for developing modules, it will also grow to include reference implementations of common modules that may be extended or otherwise altered to meet the needs of a particular federation.

For example, Figure 5 shows the "Virtual Battle Experiment – Echo" module as a version of the VMSA 3.2.2 module. The API allows this and it is strongly encouraged.

4.4 Module developer guidance

A best practice guide for developers is also provided.

5 Simulation Layer

TODO

6 FOM Library Requirements

The requirements explained in this section have come out of the experience of the Maritime Command and Control Concept Development (MC2CD) group at DRDC Atlantic.

6.1 Encoding and Decoding Support

The process of encoding attribute and parameter values to prepare to send them over an RTI, and the reverse process of decoding values that have been received are two of the most tedious and error prone aspects of federate development [1]. A FOM specific library must provide a means to make encoding and decoding as simple as possible for the developer while staying the in the problem domain of the simulation as much as possible. This will make the federate code easy to understand by a domain expert, and eliminate multiple implementations of encoding and decoding within federates that use the library.

The importance of using tried-and-tested encoding and decoding helpers is well covered in [1]. An important development in the HLA standards was the inclusion in the SISO Dynamic Link Compatible HLA API standard [3] of a requirement for RTI vendors to include encoding and decoding helpers with their implementation. This standard will be referred to as the "DLC" or "JLC" standard throughout this document.

With these points in mind, the code library must both remove the burden of encoding and decoding and make use of the encoding helpers provided with the RTI as per the DLC standard.

6.2 FOM Object and Interaction Classes

Handling HLA object and interaction classes requires two basic services:

1. Aggregation of attributes (for HLA object classes) and parameters (for HLA interaction classes) with methods for getting, setting, encoding, and decoding;

2. A facility for managing HLA handles for classes, attributes, parameters and object instances as well as relevance flags etc.

Since HLA objects are organised in an inheritance hierarchy, the code library should mimic this (true for HLA interactions as well). Modern object oriented "best practice" dictates that whenever possible designs should be oriented around interfaces (Java) or abstract classes (C++).

6.3 Developer Productivity

Developer productivity is a function of many variables, even after the personality and work habits of the individual are eliminated from the equation. Apart from these personality issues productivity is affected by training, experience, tools, the complexity of the libraries used, and the level of abstraction from the problem domain. Of these factors, the design of the library can only factor into the last three. That is, tools, library complexity, and abstraction.

The library should not abstract the developer's code too far from the underlying HLA and simulation domain concepts. We have some experience within the VCS Group with the consequences of doing this, both with our own framework for federates (Polka), and with code frameworks from third parties.

The problem with abstracting too far from HLA is that it becomes very difficult for new federate developers to understand how the framework relates to the actual HLA calls. Since HLA is not particularly hard to understand on its own the added complication of abstraction has not proved to be a benefit in our federate projects. The problem is particularly compounded when the framework API resembles the HLA API, or uses a mixture of framework and HLA API calls.

Problem domain abstraction can also be a significant problem. Generally a federate developer will be working with simulation concepts that are easily mapped into the federation's FOM. If the library hides the FOM in an abstraction layer then it increases the developer's learning curve and makes debugging of simulation issues more difficult.

Therefore the library should expose both the HLA services and FOM classes to the developer rather than hiding them in abstraction layers. In the case of a FOM library there may be little direct need to address the HLA services, but federates that implement these should try to adhere to the same principles.

Developer productivity is also enhanced when a library is IDE friendly. For example, when strong typing is used the code completion features of modern IDEs can eliminate the need for a lot of typing and the inevitable typos that come with it.

6.4 Strong Typing

More important than facilitating the use of code completion, strong typing helps avoid run-time errors and will clearly help make the library more useful. In addition to strong typing the library should minimize the need for explicit casts to data types whenever possible to reduce the amount of typing required, to help with code completion, and to avoid run-time type mismatches and casting problems.

6.5 Allowing Developer Extensions

In all but the most trivial cases, a federate that subscribes to, or publishes objects and interactions in the federation will need to add data and behaviour to those objects and interactions for internal use. For example, a simulation may need to publish a ship object in the federation for a FOM that only has state information about the ship (position, speed, orientation), but need that ship to have methods like "flank speed", "full right rudder" and so on for it's internal simulation. It could also be the case that new data fields will be needed to extend the basic FOM object and interaction classes, for example the FOM may not include the heat signature of our ship, but the federate's internal simulation requires it.

The library must allow a developer to easily add functions and data to an object or interaction without breaking the inheritance chain, or requiring copy-paste duplication of code. Code that is

generally applicable to all federates should be rolled-into the library itself and a new version created.

6.6 Code Generation

Unlike the older Distributed Interactive Simulation (DIS) standard [4] the data that is shared in an HLA federation is not part of the standard, only how that data is described and communicated. The up side is this makes HLA much more flexible than DIS; the down side is that every federation has the potential to be a “special case” with no commonality with any other.

The data definition for HLA is contained in the Federation Object Model (FOM), the format of which is part of the IEEE 1516 standard [5]. Fortunately the standard specifies an Extensible Mark-up Language (XML) schema that must be used for exchanging FOM data, and the FOM xml files can be used as the input to a code generator.

7 Library Design

This section describes the design that generated code libraries will conform to. While the design is not FOM specific, examples are included from the Real Time Platform Reference FOM version 2 (RPR2) and the Virtual Marine Systems Architecture (VMSA) FOM.

An HLA FOM consists of three basic types of information:

1. The data types that are used,
2. The object classes that can exist,
3. The interaction classes that can be sent.

There is a lot of information in a FOM, but those are the three main things that the FOM exists to describe, and are the things that a federate developer is most concerned with.

Note: "Object classes" in HLA-speak refers to entities that exist in the federated simulation, not to instantiated classes that are part of a running application.

7.1 Data types

The data types defined in the FOM are critical to defining the attributes of object classes and the parameters of interaction classes. There are several data type tables in any FOM (the tables and some of their content are specified by the IEEE standard [5]), specifically:

1. Basic data representation,
2. Simple datatype table,
3. Enumerated datatype table,
4. Array datatype table,
5. Fixed record datatype table,
6. and Variant record datatype table

Note: the misspelling of "datatype" is part of the IEEE standard and will be used in this document when referring specifically to the IEEE standard rather than the concept of a "data type".

7.2 FOM objects and interactions

Before discussing the how the library design, it is worth taking a look at how FOMs handle object and interaction hierarchies.

7.2.1 HLA object classes

HLA FOMs describe objects that may appear in a federation in a simple inheritance hierarchy, shown in Figure 6 below.

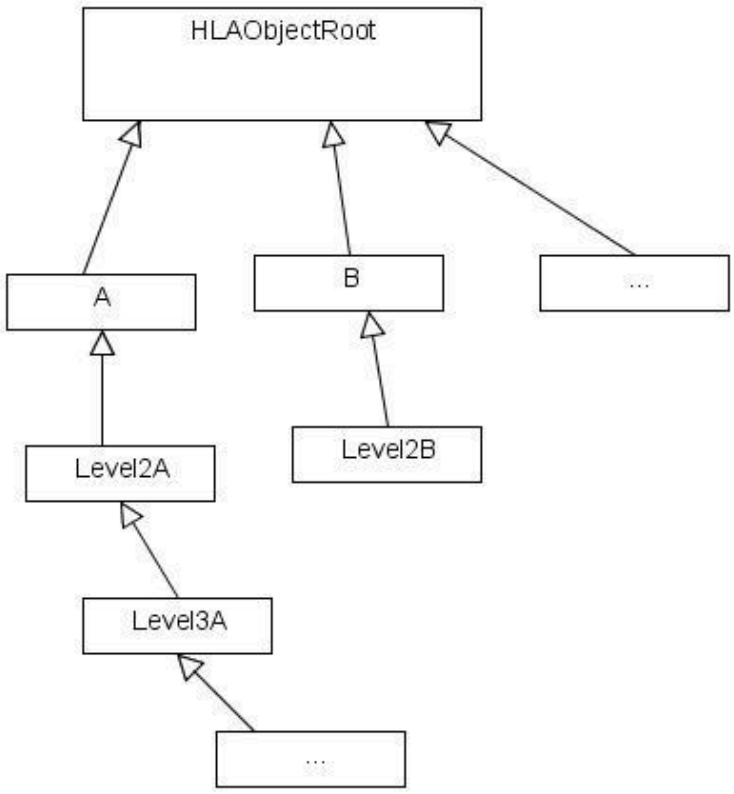

Figure 6, object class structure in HLA FOMs.

While the object classes are organised in a hierarchy, and their attributes are inherited, this is not true inheritance as a software developer understands it. Object classes do not have methods (i.e. no behaviour) and there is no polymorphism whatsoever.

Before talking about the implications for the FOM library design, it is worth looking at a concrete example. Figure 7, below, shows part of the RPR 2 Draft-18 object class hierarchy. The "Platform" class has all of the attributes of the BaseEntity class and the PhysicalEntity class as well as it's own "InteriorLightsOn" attribute.

The FOM library should mirror this hierarchy in order to keep the solution in the problem domain of the FOM. Unlike the FOM the library classes will have real inheritance, and provide methods for dealing with the attributes that belong to them.

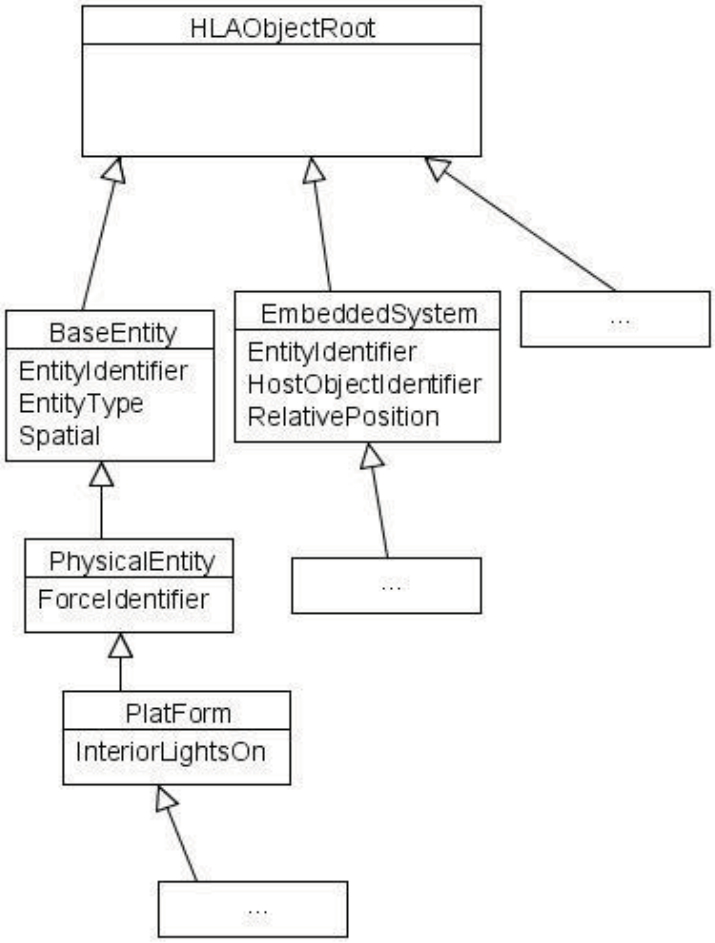

Figure 7, example hierarchy from the RPR 2.0 Draft 18 FOM.

In addition to the attributes described in the FOM, the library should also provide functionality that helps the developer work with particular instances of the FOM objects at run time. FOM objects that have been instantiated in a federation all have unique instance handles, and the federate developer will also need to know about the attribute handles and class handles that the RTI uses to indicate the "type" of object and what the data sent in an attribute update actually contains.

7.2.2 HLA interaction classes

The object and interaction hierarchies described in a FOM are very similar. In fact, as descriptions of data there is really nothing to choose between them; it is how they are handled during a federation execution that differs. FOM objects are persistent and their attributes may be updated over the course of the simulation. Interactions are transient, and any particular instance of an interaction is delivered only once to a federate.

Figure 8, example interaction hierarchy from the RPR 2.0 Draft 18 FOM. shows an piece of the RPR 2 Draft 18 FOM to illustrate the similarity between the FOM object and interaction hierarchies.

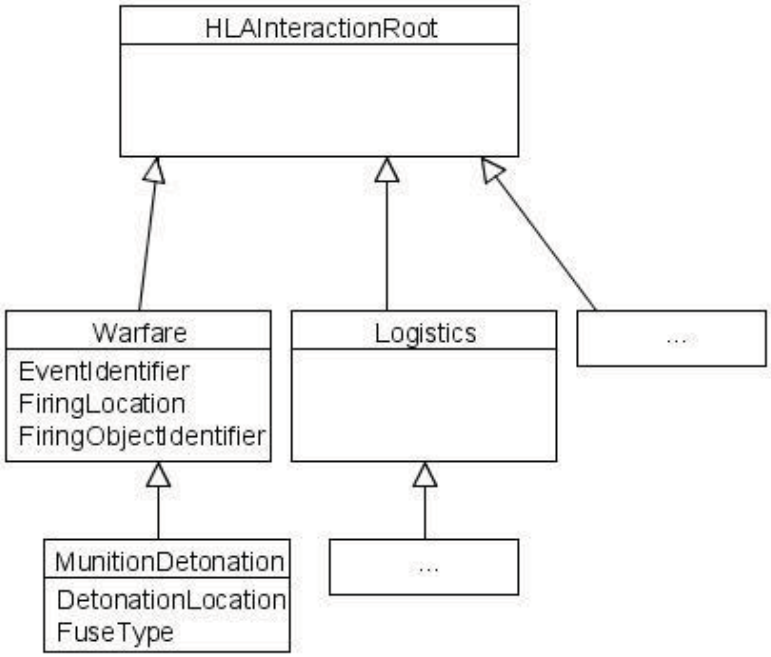

Figure 8, example interaction hierarchy from the RPR 2.0 Draft 18 FOM.

7.2.3 Design of the library

Despite the differences between objects and interactions, much of the code the library needs to deal with them is common. For example, both have a relevance flag. Because there is a need for some common functionality, both objects and interactions have a common super-class in the library design.

While it it generally good practice to “design to interfaces” in object oriented (OO) designs, in this case there is a lot of common code that would have to be replaced if the design was strictly interface based, so inheritance is a better choice. Many OO designs try to do both inheritance and interfaces along the lines of Figure 9.

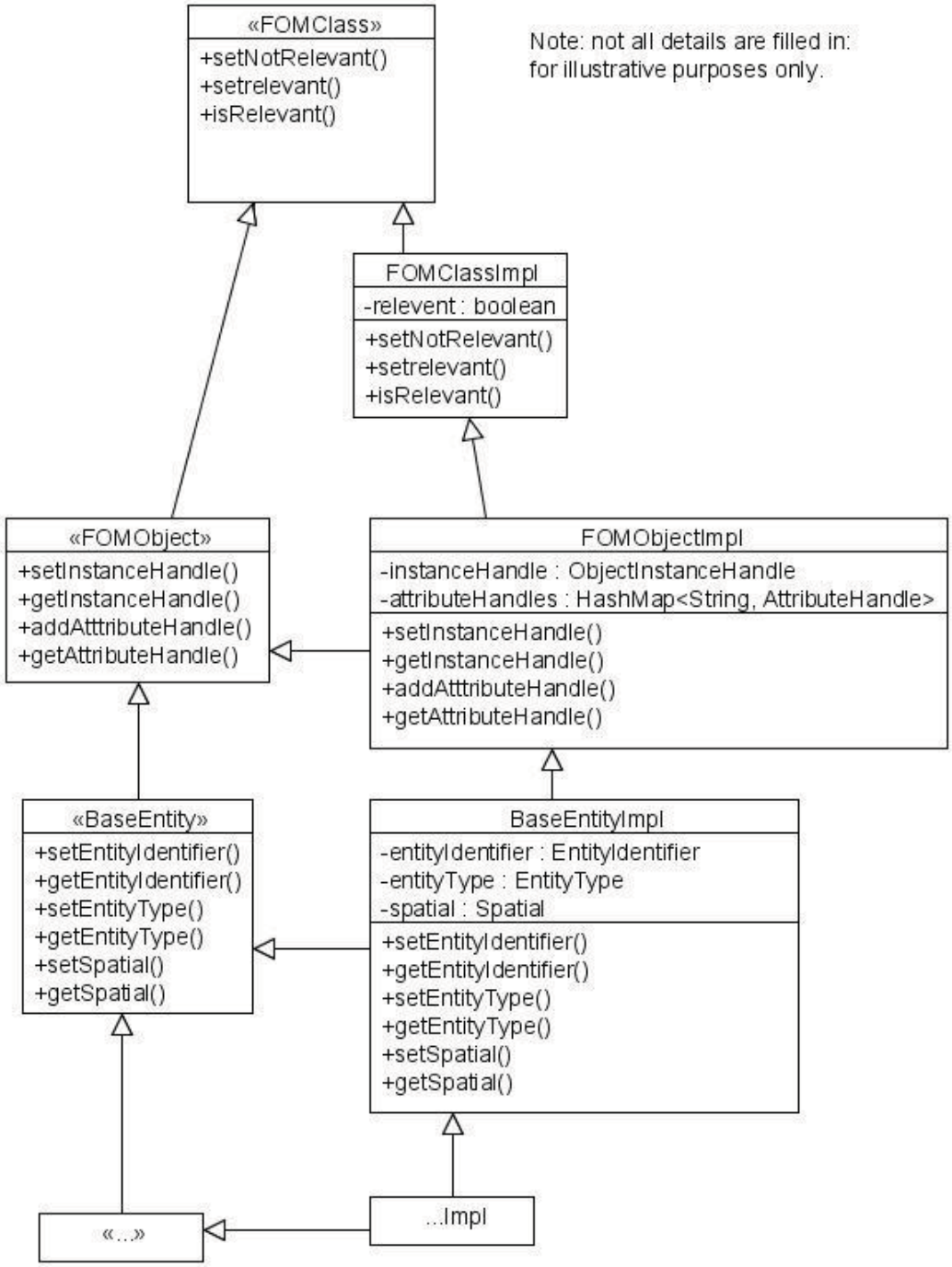

Figure 9, one possible method that combines interfaces with inheritance.

While this approach is widely used, and has it's applications, it really doesn't help us in the context of a FOM library. The addition of the inheritance tree of implementations does give us the shared behaviour that we need, but the interfaces don't really help. While it is almost certain that a federate developer would need to add federate-specific code to at least one implementation class, in doing so he would break the inheritance chain of the implementations and be forced to implement everything from his replacement class down.

To avoid breaking inheritance and still allow a developer to add functionality to the library classes, the library design uses a variation on the "Strategy Pattern". This design pattern encapsulates algorithms that can be added to other classes through aggregation. The general arrangement is shown in Figure 10.

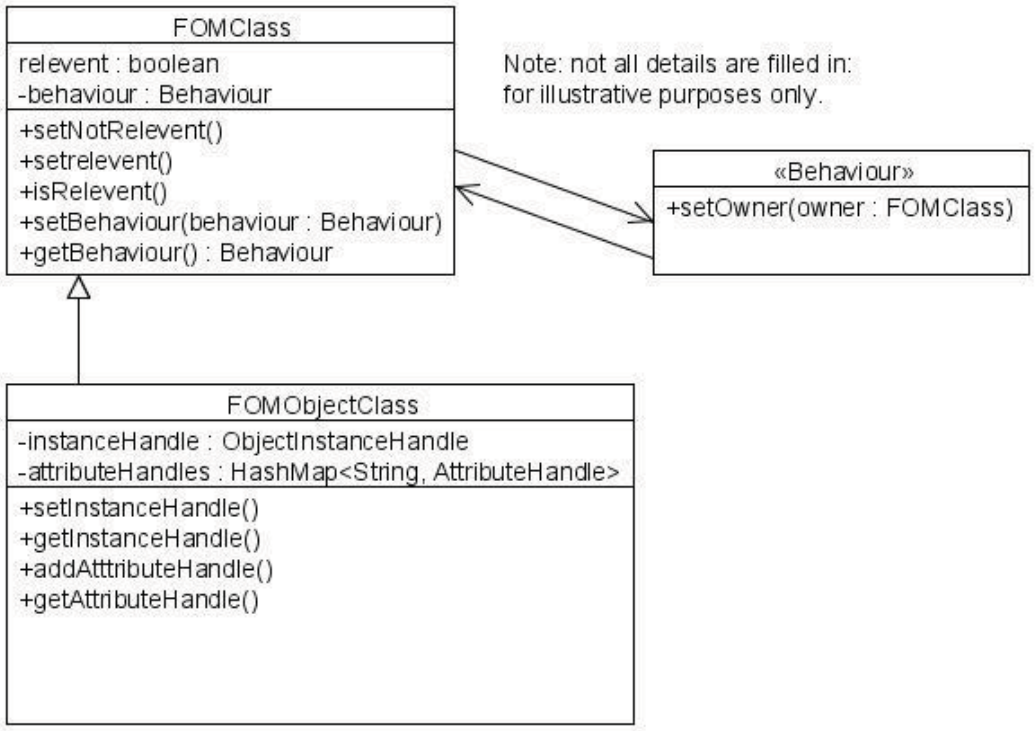

Figure 10, FOM Object class diagram showing the behaviour interface.

The "Behaviour" interface has a single method that must be implemented by concrete classes, "setOwner". The owner is the FOMClass object instance that will use the behaviour object (these are Java classes, not FOM object instance). The FOMClass includes a Behaviour variable, as well as methods to set/get the behaviour object.

Now, a developer can attach custom behaviour to a library class instance at run-time simply by creating an instance of a class that implements "Behaviour" and then calling "setBehaviour" on the library instance. The behaviour, implemented by the developer, can use the public interface of it's owner to get/set data as required. It is up to the federate developer to create these

behaviour classes and give them the functions his federate needs. For example a federate might need to simulate fighter weapons so the developer would create a "FighterWeaponBehaviour" class with "dropBomb()" and "fireMissile()" methods.

A code fragment showing how this works is shown below:

```
FighterAircraft fighter = new Fighter();
Behaviour forFighter = new FighterBehaviour();
fighter.setBehaviour(forFighter);
fighter.dropBomb();
fighter.fireMissile();
```

All library classes inherit the "setBehaviour" method of FOMClass which handles attaching the behaviour to the FOMClass object. FOMClass also sets the owner for the attached behaviour by calling it's "setOwner" method.

The "Behaviour" interface allows us to plug any object we want into a library FOMClass instance. Of course this too has it's problems, for example, in the RPR 2.0 FOM, life forms (humans, plants, whales) and fighter aircraft have a common ancestor, "PhysicalEntity". This could lead to issues where a behaviour meant for a fighter aircraft object is inadvertently attached to a whale. While bomb-dropping, missile-firing whales are a neat concept it's probably not a good idea for the library to allow this.

The library avoids this pitfall but providing an interface for every subclass of FOMObjectClass, so the "Behaviour" interface is now the top level of an inheritance hierarchy of it's own that follows the FOM object hierarchy. An example for RPR 2 Draft 18 is shown in Figure 11.

The "dropBomb()" and "fireMissile()" methods have been added to the "Air" class in the library. Of course in this example there is nothing to stop you adding this behaviour to an airliner (which is also an Air object in RPR 2) but this is a limitation of the RPR 2 FOM design.

A developer would probably want to create a class hierarchy of behaviour classes that inherit from one another as well to avoid code duplication in the behaviours, but this isn't required.

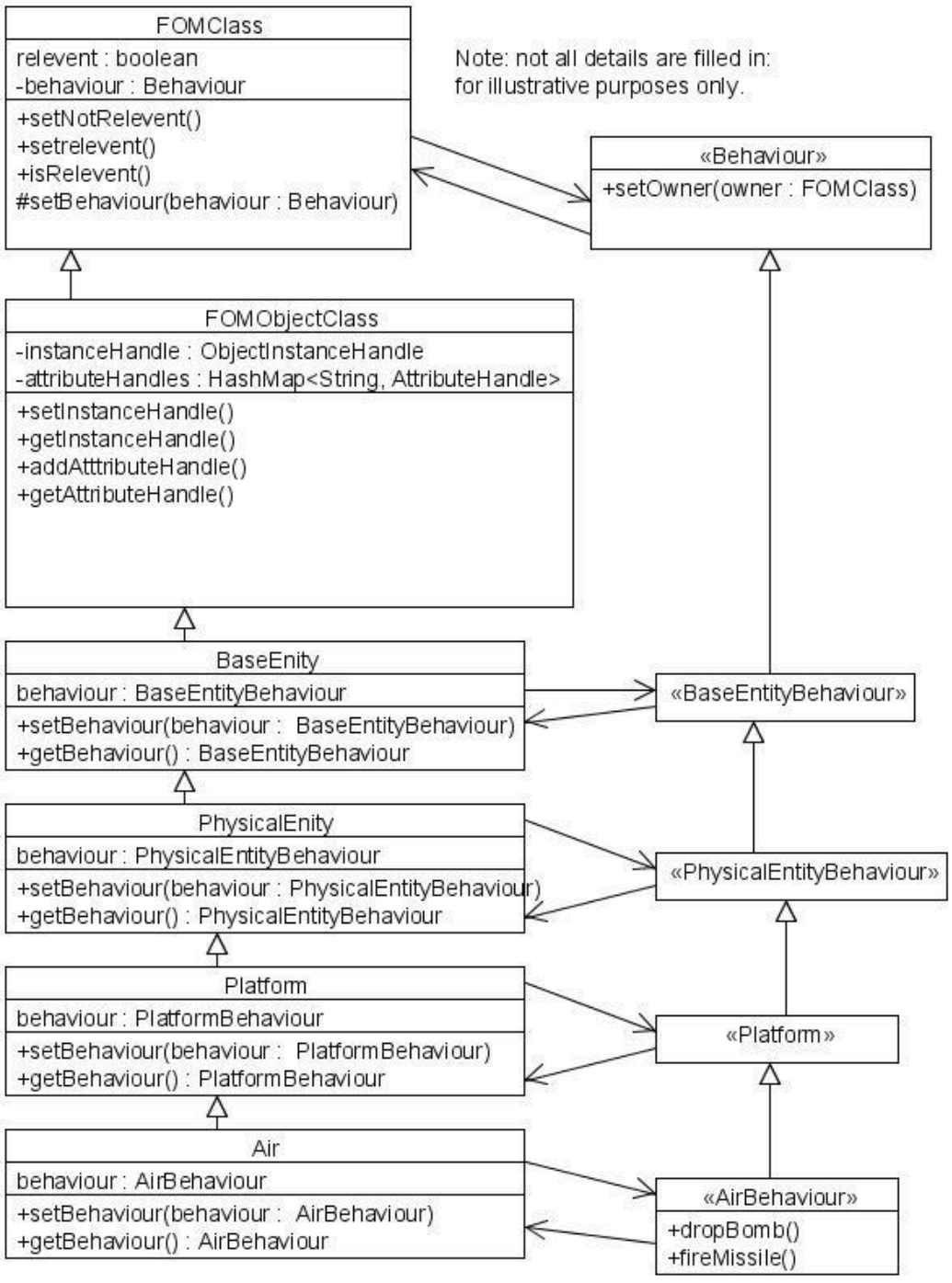

Figure 11, an example showing the Behaviour and FOMClass inheritance for a RPR 2 based FOM library.

Of course Figure 11 also shows one remaining problem with this design. Remember that this design is meant to be used to generate code libraries for a FOM that can then be reused to create any number of federates. A code generator wouldn't know that we needed "dropBomb()" and "fireMissile" methods for any particular federate project. In fact, the base library will be generated with no methods in the Behaviour interface tree other than "setOwner".

In one way this isn't really a problem. A developer just needs to create a behaviour class that implements the correct interface and then add any methods she wants to that class. The fact that an Air FOMObjectClass requires an AirBehaviour will keep her from inadvertently attaching a different type of behaviour, and she can have "dropBomb" and "fireMissile" methods in her class.

The problem comes when she want to use the methods in her behaviour. It would be nice if code completion in her IDE worked, but it won't with how things now stand. The only way code completion will work is if the AirBehaviour interface included all the methods we will need to call. To get around this problem the developer can create her own AirBehaviour interface (with the same package signature as the library) that includes the methods and then append it to the class path before the FOM library. By doing that her interface will be the one that is picked-up by the IDE so code completion will work. This also means her jar file must come before the library jar file when the federate is deployed, or the calls to her interface will cause run-time exceptions.

Replacing the library interfaces is not required in a federate that uses a FOM library based on this design, but is recommended.

7.2.4 What about FOM interactions?

The design discussion above concentrated on the FOM Object hierarchy, but everything said there applies equally to the interactions table of the FOM. In fact, this is why the Behaviour interface is used by the FOMClass rather than the FOMObjectClass in the previous figures. By placing it there the Behaviour is inherited by FOMInteractionClass and all of it's subclasses.

The library final library design is shown in XXX.

7.3 Java packages

For Java libraries the package structure helps organize code and plays a role in encapsulation rules. The library design attempts to keep classes well encapsulated, so organization and sensible code completion are the primary drivers of the package structure.

A generic Java package structure is shown in Figure 12 below, "root name" and "fom name" are meant to be parameters passed to a code generation tool. For example, if the VCS group was generating a library for the VMSA 3.2.0 FOM the full name for the "simple" package would be:

```
vcs.vmsa_322.dataTypes.simple
```

Figure 13 Shows a complete example of the library package structure for a VCS group library for the RPR 2 Draft 18 FOM. You can see from the diagram that the FOM objects and interactions also have their own packages as do library specific exceptions.

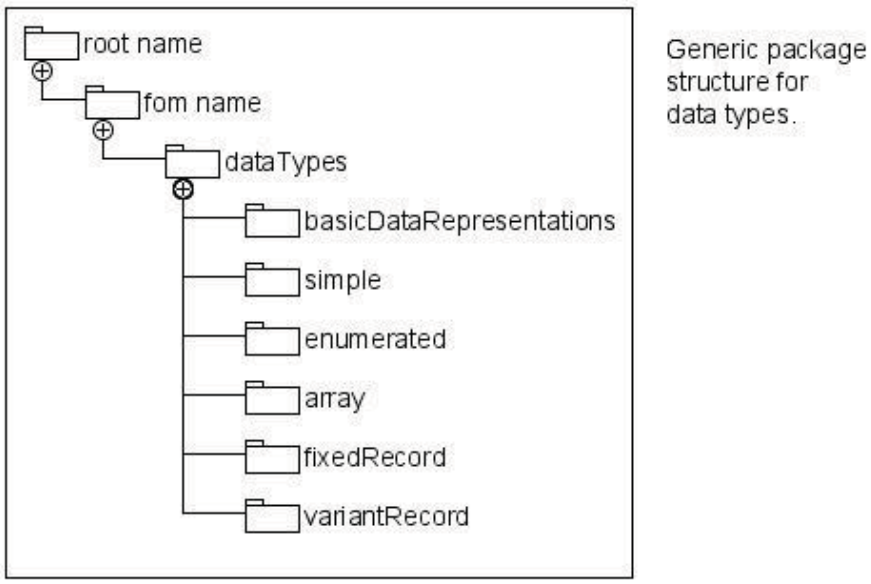

Figure 12, generic package structure of a FOM library.

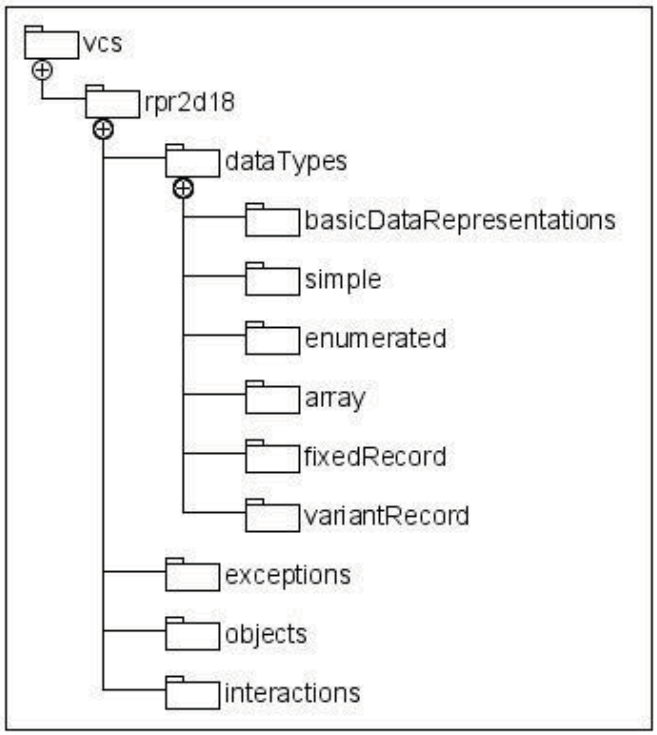

Figure 13, example package diagram for the RPR2 Draft 18 FOM showing all packages.

References

[Enter non-numbered references here]

[1] Björn Möller, Mikael Karlsson, and Björn Löfstrand*, "Reducing Integration Time and Risk with the HLA Evolved Encoding Helpers",* Proceedings of the 2006 Simulation Interoperability Workshop, siw-06s-042.pdf

[2] Allan Gillis, "Design of a Code Generator for HLA FOM libraries", …todo

[3] "*Dynamic Link Compatible HLA API Standard for the HLA Interface Spcification (IEEE 1516.1 Version)*", Simulation Interoperability Standards Organization, SISO-STD-004.1-2004, December 2004.

[4] "*IEEE Standard for Distributed Interactive Simulation*", IEEE 1278.1, 1278.1a, 1278.2, The Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York, NY 10017-2394, 1995-1998.

[5] "*IEEE Standard for Modelling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification*", IEEE 1516.2-2000, Electrical and Electronics Engineers Inc., 345 East 47th Street, New York, NY 10017-2394.

[6] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates "Head First Design Patterns", O'Reilly Media Inc, Sebastapol California, USA, 2004. ISBN 978-0-596-00712-6.

[7] "LinkedBlockingQueue", Java SE 1.6 API Documentation, Oracle Systems, http://download.oracle.com/javase/6/docs/api/

[8] "The Adaptive Communication Environment (ACE)", http://www.cs.wustl.edu/~schmidt/ACE.html

This page intentionally left blank.

Annex A [Enter Annex Level 1 heading here]

A.1 [Enter Annex Level 2 heading here]

This is an example of a paragraph. The paragraph does not say anything important, but does provide a visual of what a paragraph looks like. Below is a sample of an annex equation that has been inserted using the DRDC toolbar:

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi} \tag{A.1}$$

A.1.1 [Enter Annex Level 3 heading here]

This is an example of a paragraph. The paragraph does not say anything important, but does provide a visual of what a paragraph looks like.

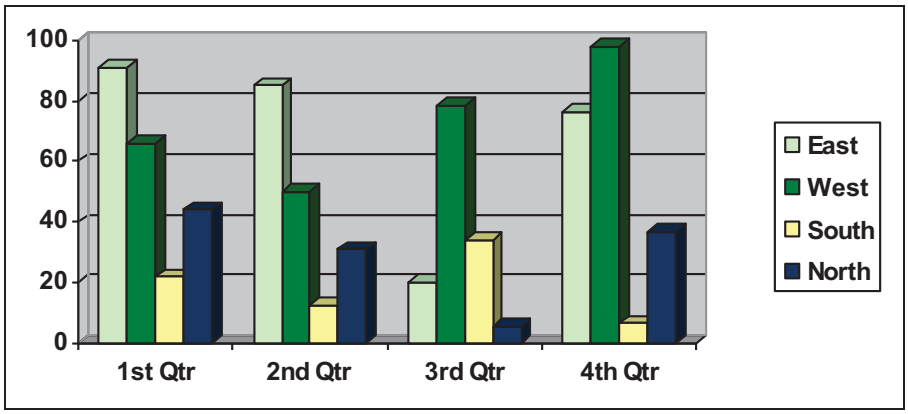

Figure A-14: This is the caption for the figure shown above.

This page intentionally left blank.

Bibliography

[Enter bibliography here if applicable. If not, delete the page.]

List of symbols/abbreviations/acronyms/initialisms

[Enter list here, if applicable. If not, delete the page.]

API	Application Programming Interface
DLC	Dynamic Link Compatible
DND	Department of National Defence
DRDC	Defence Research & Development Canada
DRDKIM	Director Research and Development Knowledge and Information Management
R&D	Research & Development
SISO	Simulation Interoperability Standards Organisation

Glossary

[Enter Technical terms and explanation of terms here, if applicable. If not, delete the page.]

Sample of a Glossary Term

The formatting of the glossary is at the discretion of the author, however this is an example of a format that is often used.

Index

[Enter index here if applicable. If not, delete the page.]

The formatting of the index is at the discretion of the author, however below is an example of a format that is often used.

V

Verso (opposite side) of title page, 28, 29, 34-35
    format, 51, 54

Distribution list

Document No.: DRDC Atlantic TM [enter number only: 9999-999]

LIST PART 1: Internal Distribution by Centre

0 TOTAL LIST PART 1

LIST PART 2: External Distribution by DRDKIM

1 Library and Archives Canada

1 TOTAL LIST PART 2

1 TOTAL COPIES REQUIRED

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – Atlantic
9 Grove Street
P.O. Box 1012
Dartmouth, Nova Scotia B2Y 3Z7

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

A Toolkit for Building RTI Independent HLA Interfaces for Simulations:

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

[Allen, J.; Lewis, A.; Smith, J.W.; von Braun, A.]

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
March 2009	41	8

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Atlantic
9 Grove Street
P.O. Box 1012
Dartmouth, Nova Scotia B2Y 3Z7

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Atlantic TM [enter number only: 9999-999]	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

[Enter text: English]

[Enter text: French]

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

[Keywords, Descriptors or Identifiers]