Copy No. _____

Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

A Toolkit for Building RTI Independent HLA Interfaces for Simulations

Allan D. Gillis
Glenn P. Franck

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2011-225
April 2014

Canada

A Toolkit for Building RTI Independent HLA Interfaces for Simulations

Allan D. Gillis
Glenn P. Franck

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2011-225
April 2014

Abstract

This document describes the design for a software framework that will make development of High Level Architecture (HLA) federates faster, and much easier for programmers new to HLA. The motivation for this comes out of our own experience here at Defence Research and Development Canada – Atlantic (DRDC Atlantic), and while supporting various units of the Canadian Armed Forces.

The design concept is multi-layered, with two fundamental modules providing a foundation for federate development. The first is an abstraction layer that appears to developers as if it were an HLA evolved compliant run time infrastructure (RTI). While not an RTI this layer translates from HLA evolved to older versions of HLA, and hides vendor implementation quirks. The second module is a Federation Object Model (FOM) library design which is meant to form the basis for code generated automatically from extensible mark-up language (XML) FOM files.

This design is the basis for the code being implemented by the open source HLAgile project on SourceForge.

Résumé

Le présent document décrit la conception d’un cadre logiciel qui accélérera le développement de fédérés d’architecture de haut niveau (HLA) et le rendra beaucoup plus facile pour les nouveaux programmeurs de la HLA. La motivation sous-jacente à cette activité est liée à notre expérience ici à Recherche et développement pour la défense Canada – Atlantique (RDDC Atlantique) et à l’appui que nous avons fourni aux diverses unités des Forces canadiennes.

La conception décrite regroupe de nombreuses couches; deux modules fondamentaux sont au cœur du développement de fédérés. Le premier est une couche d’abstraction qui apparaît aux développeurs comme une infrastructure d’exécution (RTI) conforme à une version HLA évoluée. Bien qu’il ne s’agisse pas d’une RTI, cette couche traduit l’information des versions HLA évoluées à des versions HLA antérieures tout en masquant les particularités propres à la mise en œuvre par les fournisseurs. Le second module est un concept de bibliothèque de modèles d’objets de fédération (FOM) qui vise à servir de base du code généré automatiquement à partir des fichiers FOM du langage de balisage extensible (XML).

Cette conception sert de base pour le code mis en œuvre actuellement dans le cadre du projet en source ouverte HLAgile dans SourceForge.

This page intentionally left blank.

Executive summary

A Toolkit for Building RTI Independent HLA Interfaces for Simulations

Allan D. Gillis, Glenn P. Franck; DRDC Atlantic TM 2011-225; Defence R&D Canada – Atlantic; April 2014.

Introduction: While there are a small number of competing technologies for distributed simulation, within the Canadian Forces (CF) the High Level Architecture (HLA) has been adopted as the interoperability standard. Over several years of working with distributed simulation systems, particularly those based on HLA, the authors have identified deficiencies in the tools available for software developers wishing to learn HLA or simply develop more robust HLA compatible simulations quickly.

While there are many kinds of tools which might help HLA developers our experience has shown that there are two main areas in which big gains might be made. The first is in handling the encoding and decoding of data passed via an HLA Run Time Infrastructure (RTI), and providing a code framework to handle HLA Federation Object Models (FOM). The second is in handling differences between different versions of HLA, and dealing with bugs and quirks in vendor implementations of the HLA RTI.

Significance: The proposed framework is a multi-level Application Programming Interface (API) that may be implemented in stages. Each stage is useful by itself, and together they provide a basis for faster development of more robust HLA software.

The specific benefits for the Canadian Forces and Canadian Industry in implementing the design will come from the reduction in HLA federate development costs, and an increase in the pool of HLA knowledgeable programmers generated by the provision of open source libraries. Overall this initiative is expected to reduce costs for production and maintenance of HLA software for all types of research and simulation systems.

Future plans: It is our intent to implement an open source set of tools and libraries in Java and C++ based on the design presented in this paper. Our long range intent is not to replace commercial vendor tools, but to facilitate their use and flatten the learning curve for new HLA programmers.

Sommaire

A Toolkit for Building RTI Independent HLA Interfaces for Simulations

Allan D. Gillis, Glenn P. Franck; DRDC Atlantic TM 2011-225; R & D pour la défense Canada – Atlantique; avril 2014.

Introduction : Bien qu’il existe un faible nombre de technologies concurrentes en matière de simulation répartie, l’architecture de haut niveau (HLA) a été adoptée comme norme d’interopérabilité au sein des Forces canadiennes (FC). Après avoir travaillé pendant plusieurs années à l’aide de systèmes de simulation répartie, en particulier ceux basés sur la HLA, les auteurs ont décelé des lacunes dans les outils mis à la disposition des programmeurs de logiciels qui souhaitent apprendre à utiliser la HLA ou simplement développer rapidement des simulations plus robustes qui soient compatibles HLA.

Même s’il existe de nombreux types d’outils qui pourraient aider les développeurs de la HLA, notre expérience a démontré que des gains importants pourraient être réalisés dans deux secteurs importants. Le premier est lié au traitement du codage et du décodage des données transmises par une infrastructure d’exécution (RTI) HLA, et à la constitution d’un cadre de code pour le traitement des modèles d’objets de fédération (FOM) HLA. Le second concerne le traitement des écarts entre les différentes versions de la HLA et la gestion des bogues et des particularités propres à la mise en œuvre par les fournisseurs de la RTI HLA.

Importance : Le cadre proposé est une interface de programmation d’applications (API) regroupant de nombreuses couches et pouvant être mise en œuvre en étapes. Chacune d’elles est utile en soi et ensemble elles sont au cœur d’un développement plus rapide de logiciels HLA plus robustes.

Cette conception et la mise en œuvre connexe offriront aux Forces canadiennes et à l’industrie canadienne des avantages précis, à savoir la réduction des coûts du développement des fédérés de HLA et une augmentation du bassin de programmeurs compétents en HLA grâce à l’utilisation de bibliothèques en source ouverte. Globalement, on s’attend à ce que cette initiative entraîne une réduction des coûts de production et de maintenance des logiciels HLA pour tous les types de systèmes de recherche et de simulation.

Plans futurs : Notre intention est de mettre en œuvre un ensemble d’outils et de bibliothèques en source ouverte en Java et C++ d’après la conception présentée dans le présent document. À long terme, notre intention n’est pas de remplacer les outils offerts par les fournisseurs dans le commerce, mais de faciliter leur emploi tout en aplanissant la courbe d’apprentissage pour les nouveaux programmeurs en HLA.

Table of contents

List of figures

This page intentionally left blank.

1 Introduction

The High Level Architecture (HLA) for connecting distributed simulations has been in use since 1998 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, and then 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 was recently approved as an Institute of Electrical and Electronics Engineers (IEEE) standard. The two standards-based versions of HLA are IEEE 1516.2000 with its various amendments, including the Simulation Interoperability Standards Organisation (SISO) Dynamic Link Compatible (DLC) API standard [1] and IEEE 1515.2010 (also known as HLA evolved, or 1516e). Previous to IEEE 1516 there were many variations of HLA all derived from the Defence Modelling & Simulation Office (DMSO) versions and collectively referred to as the "1.3" version of HLA. Unfortunately while these various 1.3 Runtime Infrastructures (RTI) 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.

In addition to writing a simulation and understanding multiple versions of the HLA, a developer 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. Every use of a FOM is usually accompanied by a federation agreement (formal or informal) that further specifies how certain situations important to the overall system shall be handled, and that goes beyond the basic definitions of objects and attributes.

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 Maritime Command & Control Concept Development (MC2CD) group, and of others [2], that encoding and decoding the values of attributes and 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 its 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 developers meet the twin challenges of multiple RTI versions and FOMs while allowing them 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.

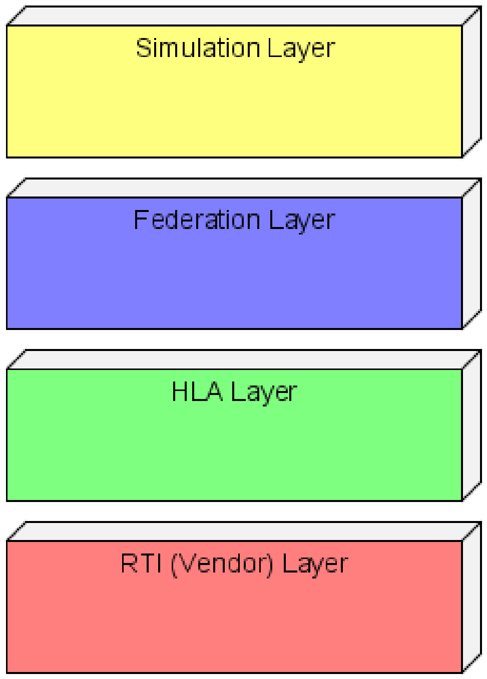

Figure 1: Overall architecture.

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 HLA Evolved (1516.2010) standard. 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 components needed to actually join a federation using a particular FOM and federation agreement. Potentially there will be a separate federation layer component for every FOM / federation agreement combination. For example, the Real Time Platform Reference (RPR) FOM version 1 [3] would have its own set of Federation Layer Components, as would the RPR FOM version 2.0 (RPR2) [4].

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 the one we have no control over. Vendors will do what they do for their own reasons, whether open source projects or commercial companies. Changes to 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 Distributed Interactive Simulation (DIS) [5] or the Test and Training Enabling Architecture (TENA) [6]. 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 Virtual Maritime Systems Architecture [7] and RPR2 based FOMs meaning we may only need two basic modules that can be subclassed to suit 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 its 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 that use 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 [8] found in java.util.concurrent. C++ implementations shall utilise a cross platform open source library that includes a similar capability (ACE [9] for example).

3 HLA Abstraction Layer (HAL)

As outlined in Section 2.1.1, 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.

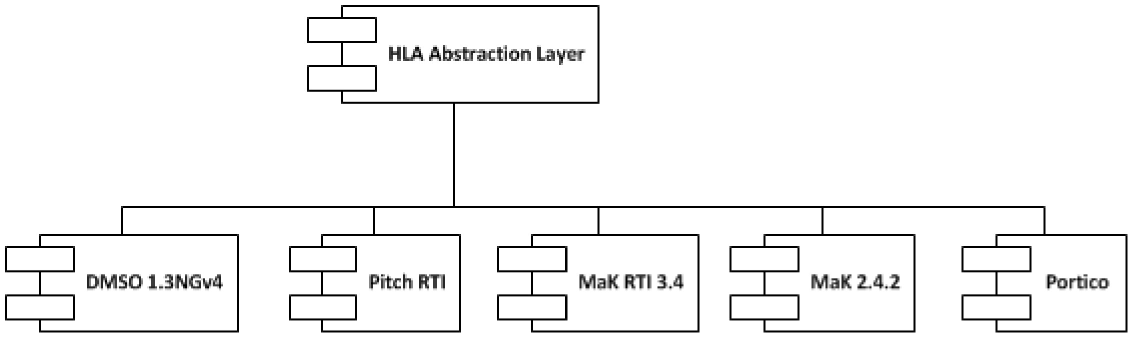

Figure 2: How the HLA Abstraction Layer hides vendor implementations.

In Figure 2, five 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 [1], IEEE 1516 standard [12] 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.

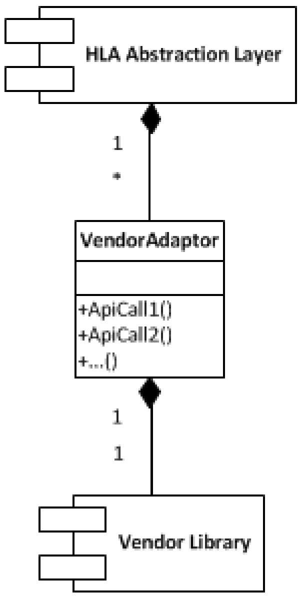

Figure 3: Adapting vendor libraries to the HLA Abstraction Layer.

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

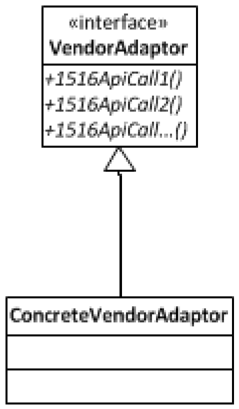

Figure 4: VendorAdaptor interface.

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 (FEL)

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.

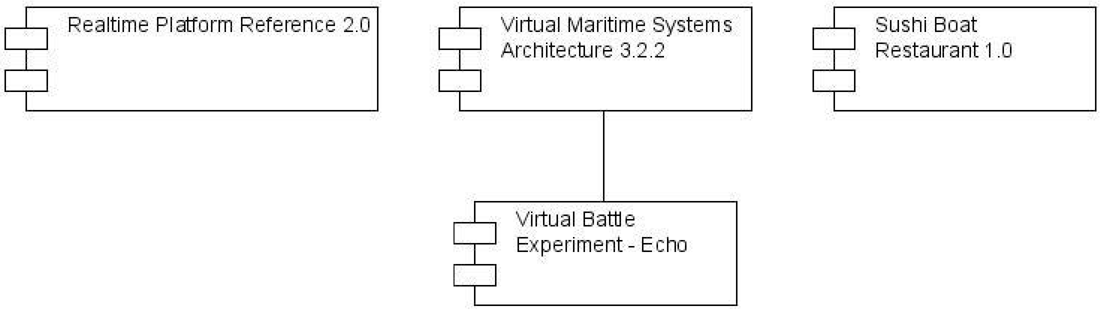
Realtime Platform Reference 2.0
Virtual Maritime Systems Architecture 3.2.2
Sushi Boat Restaurant 1.0
Virtual Battle Experiment - Echo

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 its accompanying federation agreement. For example, a RPR federation layer should use the RPR FOM and behave as specified in the RPR guidance document (often referred to as GRIM RPR) [13].

Given the flexible nature of HLA, it is entirely possible that no two federations will adhere to the same combination of FOM and guidance (also known as a Federation Agreement). However, creation of an entirely new FOM requires significant effort for all but trivial cases, and in our experience federation agreements are normally variations on previous ones used for similar purposes. As a result, Federation Encapsulation implementations must be designed with extensibility and future modification in mind.

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

4.3 Layer API

At this time no fixed design is envisioned for the FEL. Any FEL implementation is required to make use of FOM libraries following the design laid out in Section 7 and the HAL as described in Section 3. Until experience has been gained with implementations of this design, no further work on specification of a common FEL API is envisioned.

A best practice guide for developers is also envisioned for this project, though not yet written.

5 Simulation Layer (SL)

The purpose of the SL layer is to allow reuse of FEL modules by de-coupling the FEL module from any particular simulation system. The idea is that a FEL module for a particular federation could be built once and then used for a sonar simulation, a flight simulator, a logistics simulation, or any other federate that uses the same FOM and federation agreement.

Although this layer is envisioned as part of the overall design, work on requirements, detailed design, and implementations will not proceed until experience has been gained with the FEL and HAL.

6 FOM Library Requirements

The requirements explained in this section have come out of the experience of the 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 [2]. A FOM specific library must provide a means to make encoding and decoding as simple as possible for the developer while staying 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 [2]. An important development in the HLA standards was the inclusion in the SISO Dynamic Link Compatible HLA API standard [1] 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 MC2CD 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 Integrated Development Environment (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 its 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 [11] 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 little or 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 [12]. 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 FOM 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 [12]), 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 how the library is designed, 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.

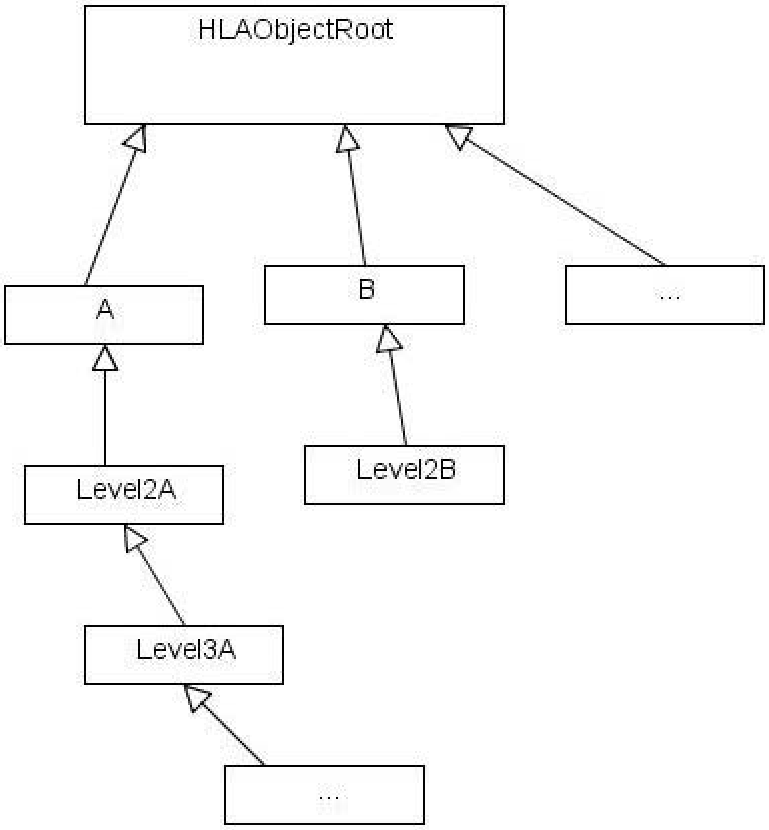

Figure 6: Object class structure in HLA FOMs.

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 also 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 its 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.

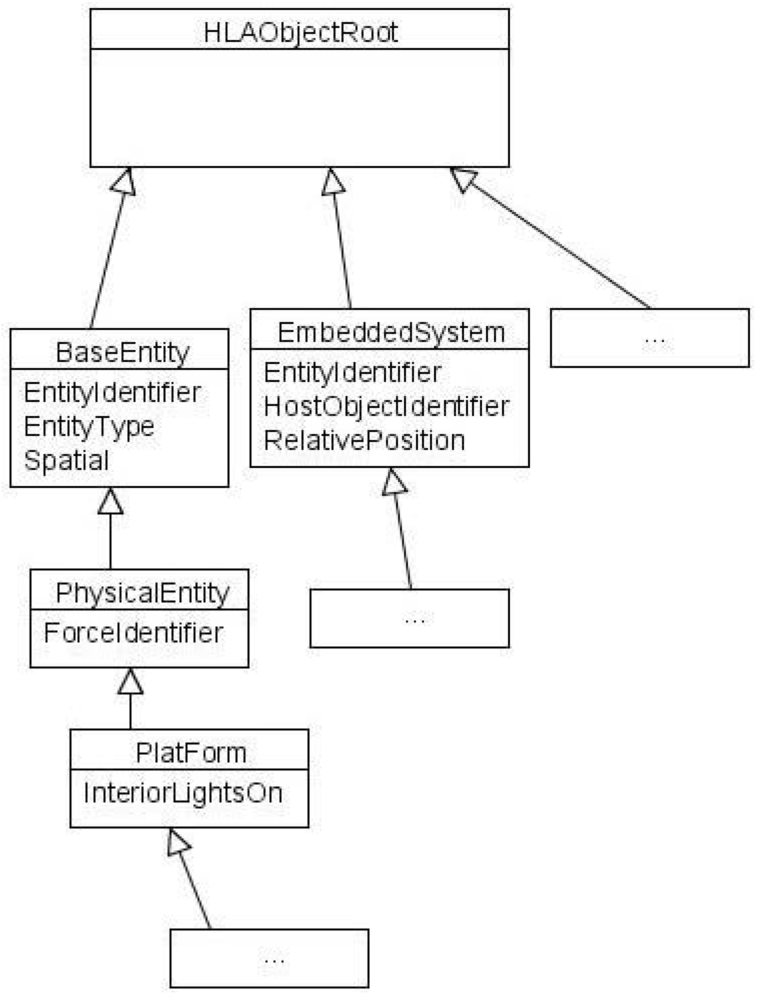

Figure 7: Example hierarchy from the RPR 2.0 Draft 18 FOM.

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 shows a piece of the RPR 2 Draft 18 FOM to illustrate the similarity between the FOM object and interaction hierarchies.

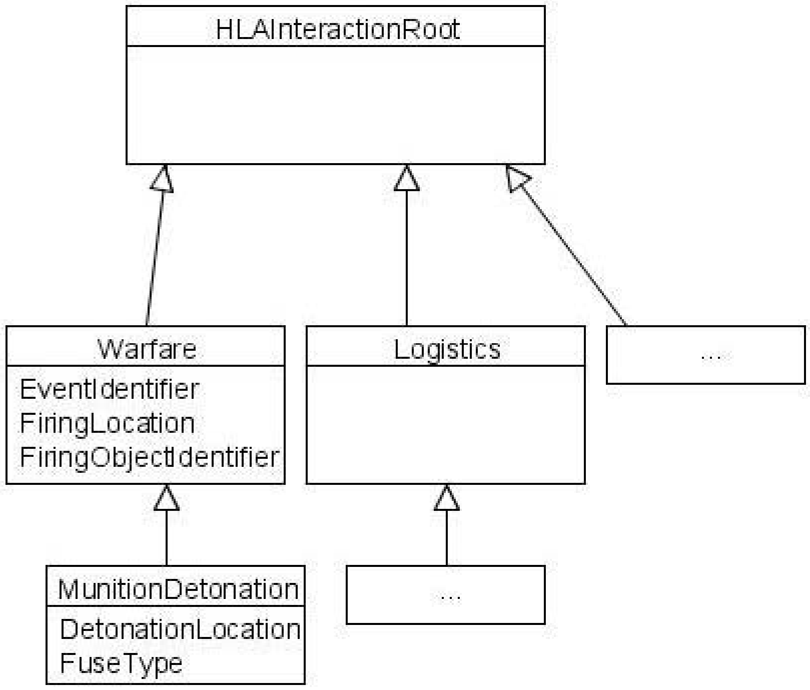

Figure 8: Example interaction hierarchy from the RPR 2.0 Draft 18 FOM.

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. Since there is a need for some common functionality, both objects and interactions have a common super-class in the library design.

While it is 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.

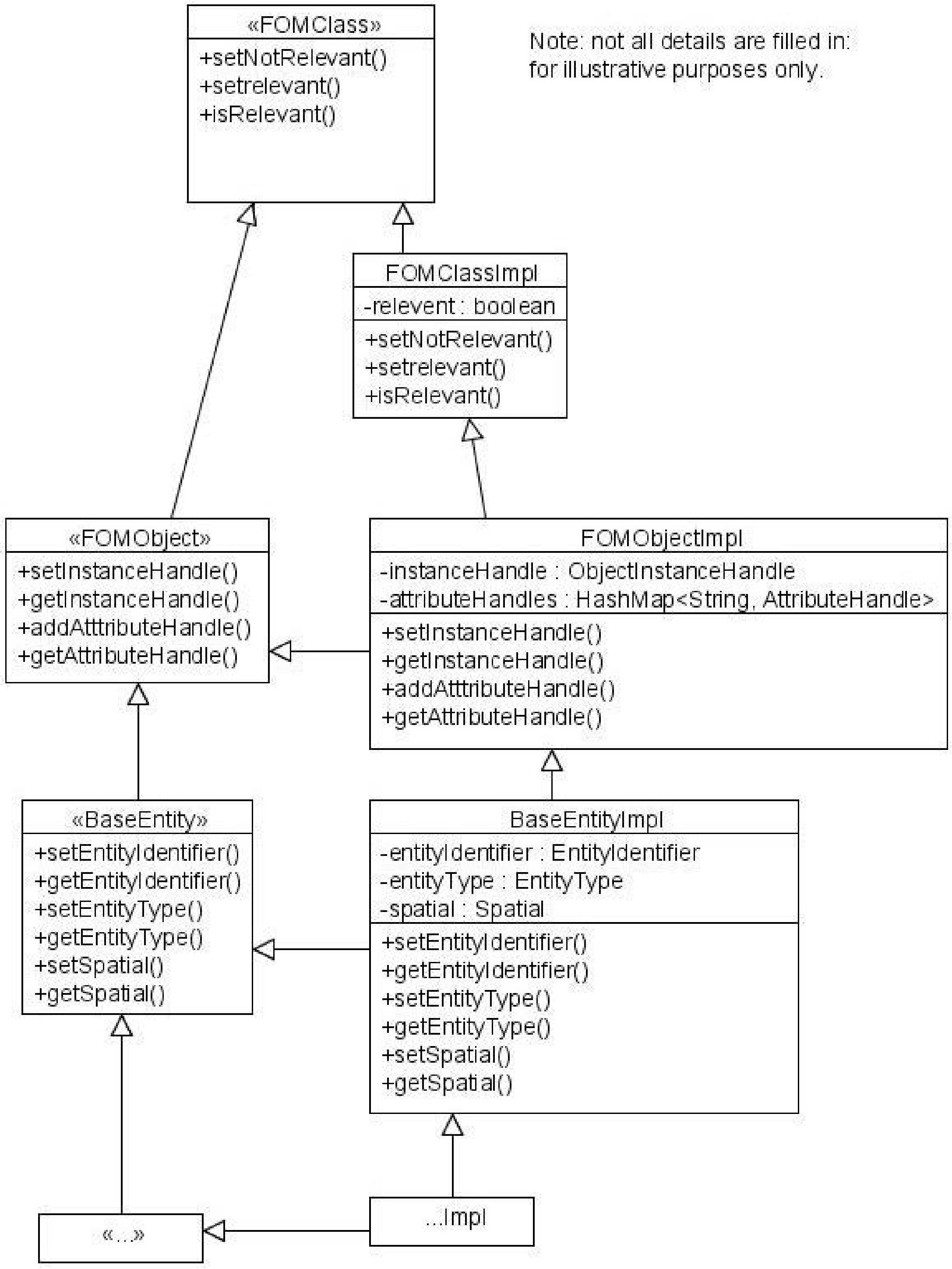

Figure 9: One possible method that combines interfaces with inheritance.

While this approach is widely used, and has its 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” [14]. This design pattern encapsulates algorithms that can be added to other classes through aggregation. The general arrangement is shown in Figure 10.

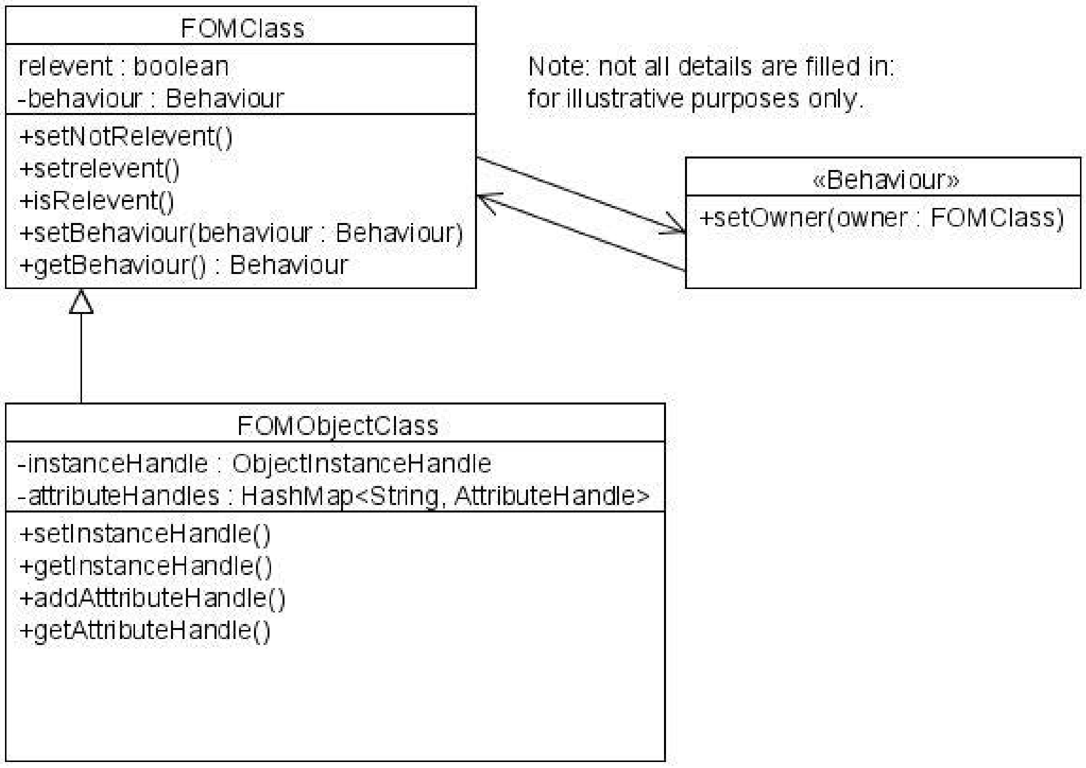
FOMClass
relevent : boolean
-behaviour : Behaviour
+setNotRelevent()
+setrelevent()
+isRelevent()
+setBehaviour(behaviour : Behaviour)
+getBehaviour() : Behaviour
Note: not all details are filled in:
for illustrative purposes only.
«Behaviour»
+setOwner(owner : FOMClass)
FOMObjectClass
-instanceHandle : ObjectInstanceHandle
-attributeHandles : HashMap<String, AttributeHandle>
+setInstanceHandle()
+getInstanceHandle()
+addAtttributeHandle()
+getAttributeHandle()

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 instances). 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 its 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 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 its "setOwner" method.

The "Behaviour" interface allows us to plug any object we want into a library FOMClass instance. Of course this too has its 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 is 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 its 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.

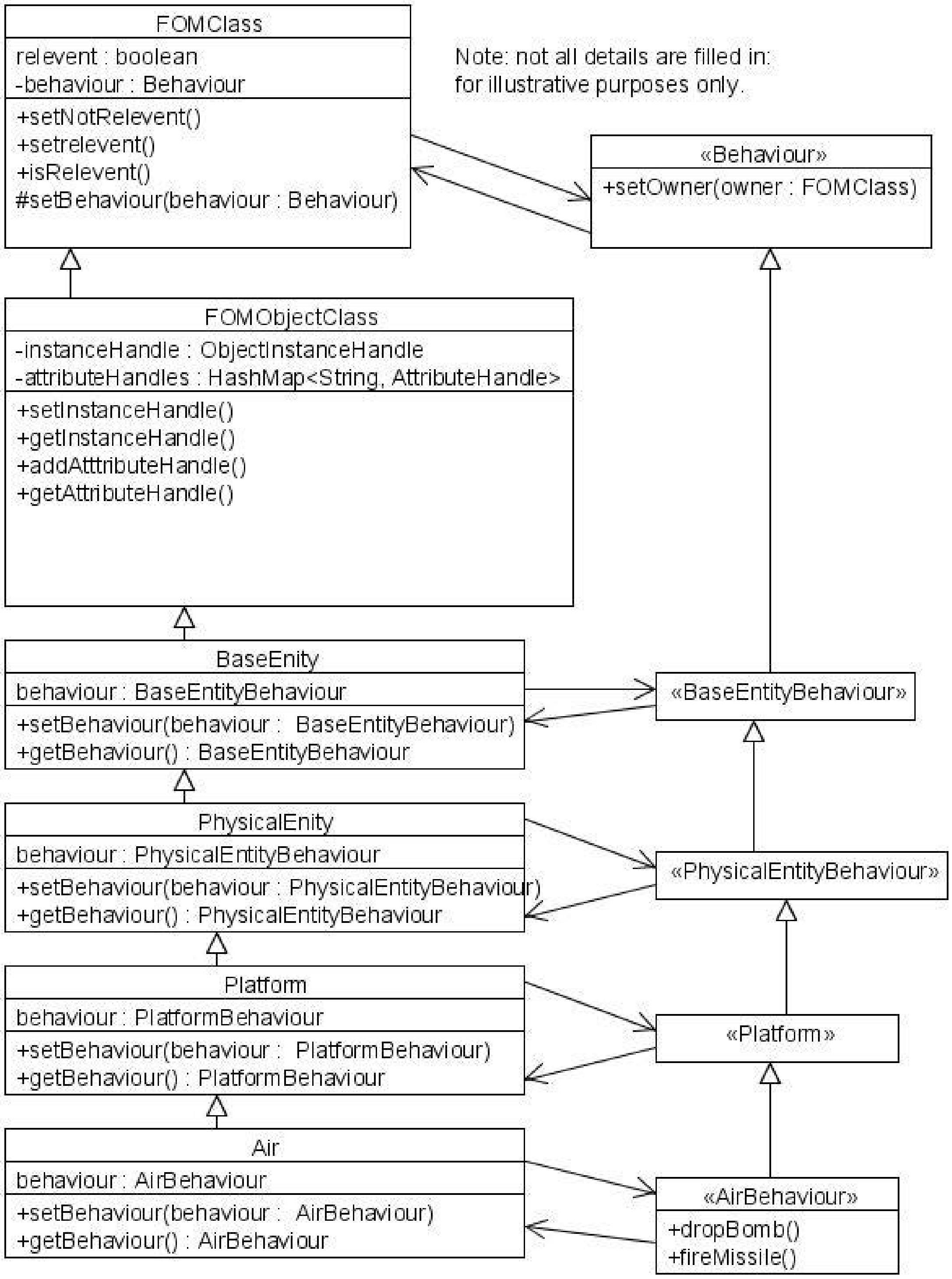
FOMClass
relevent : boolean
-behaviour : Behaviour
+setNotRelevent()
+setrelevent()
+isRelevent()
#setBehaviour(behaviour : Behaviour)
Note: not all details are filled in:
for illustrative purposes only.
«Behaviour»
+setOwner(owner : FOMClass)
FOMObjectClass
-instanceHandle : ObjectInstanceHandle
-attributeHandles : HashMap<String, AttributeHandle>
+setInstanceHandle()
+getInstanceHandle()
+addAtttributeHandle()
+getAttributeHandle()
BaseEnity
behaviour : BaseEntityBehaviour
+setBehaviour(behaviour : BaseEntityBehaviour)
+getBehaviour() : BaseEntityBehaviour
«BaseEntityBehaviour»
PhysicalEnity
behaviour : PhysicalEntityBehaviour
+setBehaviour(behaviour : PhysicalEntityBehaviour)
+getBehaviour() : PhysicalEntityBehaviour
«PhysicalEntityBehaviour»
Platform
behaviour : PlatformBehaviour
+setBehaviour(behaviour : PlatformBehaviour)
+getBehaviour() : PlatformBehaviour
«Platform»
Air
behaviour : AirBehaviour
+setBehaviour(behaviour : AirBehaviour)
+getBehaviour() : AirBehaviour
«AirBehaviour»
+dropBomb()
+fireMissile()

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 using the methods in a behaviour. It would be nice if code completion in the 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 that could be called

One way to get around this problem is for developers to create their own AirBehaviour interface (with the same package signature as the library) including all the methods and then append it to the Java CLASSPATH before the FOM library. By doing that, the interface will be the one that is picked-up by the IDE so code completion will work. This also means the developer jar file must come before the library jar file when the federate is deployed, or the calls to the interface will cause run-time exceptions.

Unfortunately, in our experience, that approach causes big problems during system integration, and when building a releasable federate. Errors creep into the CLASSPATH and result in difficult to trace problems at run-time.

The solution is to have the code generator create a set of Java files for the behaviour hierarchy, but to ***not*** include these in the executable library code. Instead two jar files will be created for the library, one containing the blank behaviours, and another containing the rest of the library. In addition a zip file containing the source of the behaviour jar will be included in the distribution package.

Now, developers have the blank behaviour source as a template for building their own classes, a jar file of blanks that will allow everything to compile, and an easy way to substitute their own behaviour library into a final product; simply replace the blank jar file with their own.

Replacing the behaviour 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 its subclasses.

The top level of the inheritance tree is shown in Figure 12.

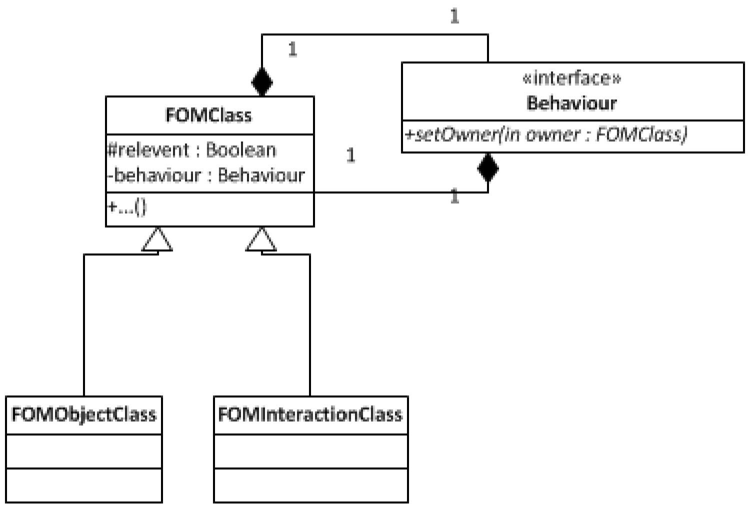

Figure 12: Top level library class diagram.

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 13below, “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
```

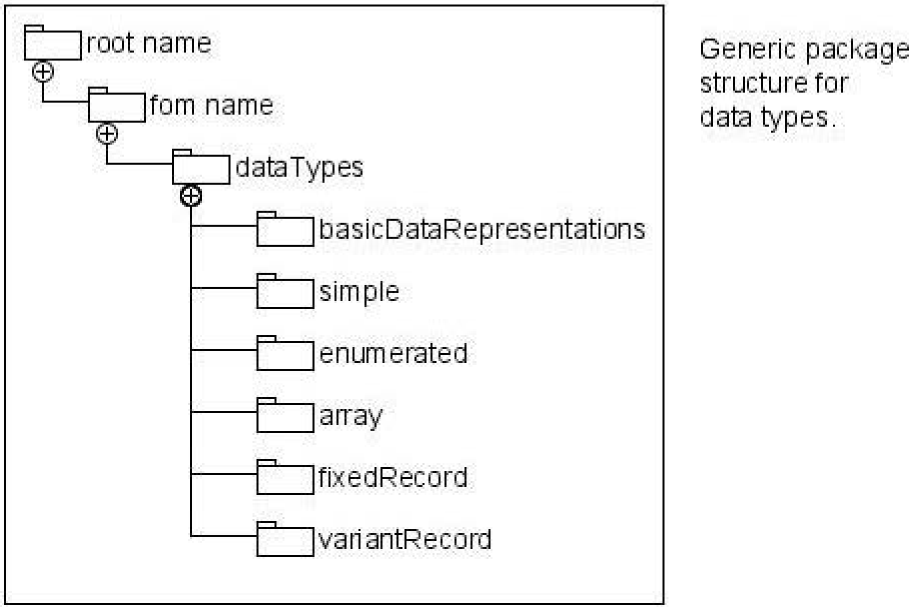

Figure 13: Generic package structure of a FOM library.

Figure 14 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.

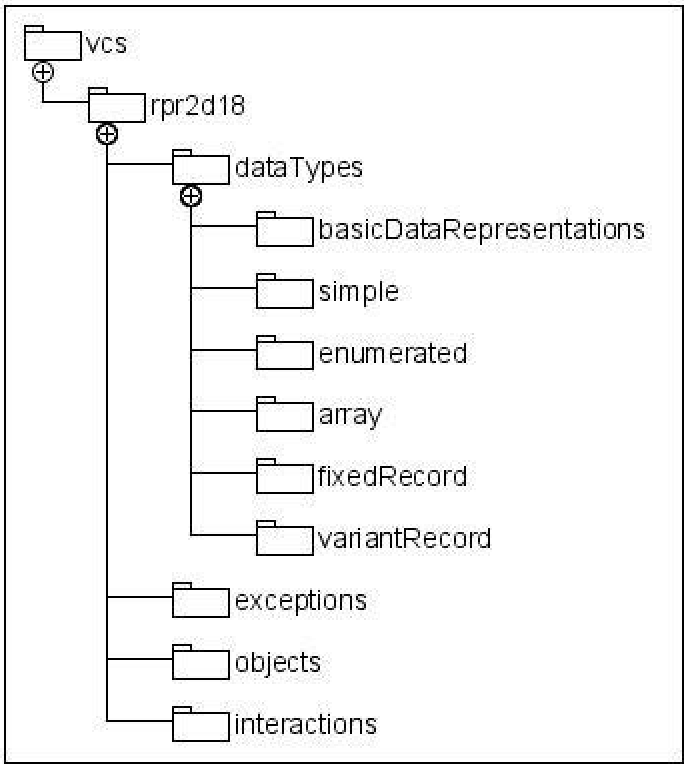

Figure 14: Example package diagram for the RPR2 Draft 18 FOM showing all packages.

References

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

[2] 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.

[3] "RPR-FOM Version 1.0", SISO-STD-001.1-1999, Simulation Interoperability Standards Organisation, 1999. www.sisostds.org.

[4] "RPR-FOM Version 2.0 Draft 17", Simulation Interoperability Standards Organisation, 1999. www.sisostds.org.

[5] "IEEE Standard for Distributed Interactive Simulation", The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, NY, 10017-2394 USA, 1995.

[6] "Test and Training Enabling Architecture (TENA)", https://www.tena-sda.org.

[7] Canney, Shane A., "Virtual Maritime System Architecture Description Document Issue 2.00", Defence Science and Technology Organisation, Edinburgh, South Australia, Australia, 2002.

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

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

[10] Allan Gillis, "Design of a Code Generator for HLA FOM libraries", unpublished DRDC Atlantic report.

[11] "*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.

[12] "*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.

[13] Reilly, Sean and Briggs, Keith, "Guidance, Rational, and Interoperability Modalities for the Real-time Platform Reference Federation Object Model (RPR FOM)", Simulation Interoperability Standards Organisation, 1999. www.sisostds.org.

[14] 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.

This page intentionally left blank.

List of symbols/abbreviations/acronyms/initialisms

API	Application Programming Interface
DIS	Distributed Interactive Simulation
DLC	Dynamic Link Compatible
DMSO	Defence Modelling and Simulation Office
DND	Department of National Defence
DRDC	Defence Research & Development Canada
FOM	Federation Object Model
HLA	High Level Architecture
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineers
R&D	Research & Development
RPR	Real Time Platform Reference (a FOM)
RPR2	RPR version 2
RTI	HLA Run-Time Infrastructure
SISO	Simulation Interoperability Standards Organisation
TENA	Test and Training Enabling Architecture

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 (NON-CONTROLLED GOODS) DMC A REVIEW: GCEC APRIL 2011

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)

Gillis, A.D.; Franck, G.P.

5. DATE OF PUBLICATION (Month and year of publication of document.) April 2014	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) 40	6b. NO. OF REFS (Total cited in document.) 14

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.) 11BZ	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.) DRDC Atlantic TM 2011-225	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

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

Unlimited

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.))

Unlimited

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.)

This document describes the design for a software framework that will make development of High Level Architecture (HLA) federates faster, and much easier for programmers new to HLA. The motivation for this comes out of our own experience here at Defence Research and Development Canada – Atlantic (DRDC Atlantic), and while supporting various units of the Canadian Armed Forces.

The design concept is multi-layered, with two fundamental modules providing a foundation for federate development. The first is an abstraction layer that appears to developers as if it were an HLA evolved compliant run time infrastructure (RTI). While not an RTI this layer translates from HLA evolved to older versions of HLA, and hides vendor implementation quirks. The second module is a Federation Object Model (FOM) library design which is meant to form the basis for code generated automatically from extensible mark-up language (XML) FOM files.

This design is the basis for the code being implemented by the open source HLAgile project on SourceForge.

Le présent document décrit la conception d'un cadre logiciel qui accélérera le développement de fédérés d'architecture de haut niveau (HLA) et le rendra beaucoup plus facile pour les nouveaux programmeurs de la HLA. La motivation sous-jacente à cette activité est liée à notre expérience ici à Recherche et développement pour la défense Canada – Atlantique (RDDC Atlantique) et à l'appui que nous avons fourni aux diverses unités des Forces canadiennes.

La conception décrite regroupe de nombreuses couches; deux modules fondamentaux sont au cœur du développement de fédérés. Le premier est une couche d'abstraction qui apparaît aux développeurs comme une infrastructure d'exécution (RTI) conforme à une version HLA évoluée. Bien qu'il ne s'agisse pas d'une RTI, cette couche traduit l'information des versions HLA évoluées à des versions HLA antérieures tout en masquant les particularités propres à la mise en œuvre par les fournisseurs. Le second module est un concept de bibliothèque de modèles d'objets de fédération (FOM) qui vise à servir de base du code généré automatiquement à partir des fichiers FOM du langage de balisage extensible (XML).

Cette conception sert de base pour le code mis en œuvre actuellement dans le cadre du projet en source ouverte HLAgile dans SourceForge.

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.)

High Level Architecture, Middleware, Software Library, Software Architecture, Toolkit

This page intentionally left blank.