Solving the FOM-Independence Problem

Len Granowetter
MAK Technologies, Inc.
185 Alewife Brook Parkway
CAMBRIDGE, MA 02138
lengrano@mak.com
http://www.mak.com

Keywords:
HLA, FOM-Agile, Middleware

ABSTRACT: One of the key concerns facing developers of HLA simulation applications, is that they need to be able reuse federates in different federation executions, each with its own way of representing data in a FOM. This paper presents and contrasts two predominant methods that help to solve this FOM-independence problem. The first approach involves using a code generator to generate a FOM-specific middleware API, which necessitates changing application code when data representation changes. By contrast, a FOM-configurable, fixed-API middleware toolkit can enable federates to switch among different FOMs without changing application code.

1. Introduction

One of the most beneficial features of the HLA, is that it allows the developers of each federation to decide how the federation's data should be represented in its Federation Object Model (FOM). The same simulation concepts might be represented differently in different FOMs because of differences in bandwidth/accuracy tradeoffs in the federations, because the some key federates in a federation use a particular data representation internally, or simply because different FOMs have evolved from different previous-generation data standards.

Wouldn't it be nice if you could build a simulation application once, and have it automatically be able to play in any federation, using any arbitrary FOM (or at least any of a set of FOMs that represents the concepts needed by a particular simulation domain)? Many believe that this is a key promise of HLA, but in practice, no one has really gotten there yet. A question that we are often asked is why?

Many different organizations, MAK Technologies included, have developed approaches and middleware tools that make it easier for applications to switch among different FOMs. Some have used the term "FOM-Agility" to refer to the ability to switch FOMs fairly easily. We will discuss two predominant approaches, and explain which approach we think is most applicable to which situations.

2. The Role of Middleware

Most people that have been involved in developing HLA applications are familiar with the concept of middleware. Simply put, a middleware library implements a set of functionality above the level of the RTI, that is required by all HLA applications, or at least by all applications in a particular federation execution.

It is not necessary that applications use a middleware layer, but if they do not, the effort required to implement this functionality must be duplicated for each application. When middleware is used, this functionality is implemented once, and is reused by each application through library calls. Middleware is so named because of its position between application code and the RTI.

Two of an application's responsibilities that are often implemented in middleware are:

Maintaining a database of remotely simulated objects and their current states. The RTI does not maintain a database of objects that can be queried for current attribute values by an application. It is simply the communications mechanism through which messages describing object creations, removals and attribute updates are exchanged among federates.

It is up to the application to maintain such a database, adding new objects in response to the discoverObjectInstance RTI service, removing them in response to the removeObjectInstance RTI service, and updating components of their current state in response to attribute updates delivered by the reflectAttributeValues RTI service. (Attribute updates typically contain values for only a subset of an object's attributes, rather than its entire state.)

When this functionality is in a middleware layer, an application can just query its database to obtain the current state of each remote object, ignoring the details of which attributes have been updated when.

Managing the transmission of attribute updates for locally simulated objects. The RTI does not keep track of the current state of your locally simulated objects either, so it can neither automatically know when attribute values are out of date and thus need to be communicated to other federates, nor automatically build and send attribute updates.

The RTI also does not maintain a database that can be queried by an application to find out which object classes

and attributes are currently subscribed to by other federates (and thus should be registered by the application and updated whenever update conditions are true), and which attributes have recently been requested by other federates (and thus need to be sent now even if not out of date).

It is up to the application to manage these databases, in response to RTI-initiated service invocations like start/stopRegistrationForObjectClass, turnUpdatesOn/OffForObjectInstance, and provideAttributeValueUpdate.

When this functionality is implemented in a middleware layer, the application simply needs to set the current state of locally simulated objects each frame, without worrying about what needs to get sent when. The middleware layer figures this out and takes care of building and sending the updates through the RTI.

3. Middleware and FOM-Independence

Most middleware developers would agree with the descriptions and definitions above, whether they are developing libraries merely to be used by a few applications in a single federation execution, or to be sold as a commercial product and used in many different federation executions.

However, differences in approaches appear as we consider the issue of FOM-independence. Questions we can ask about various approaches include: How tied to a particular FOM is the API? Is there an infrastructure in place such that the API does not need to change when the FOM changes, or when switching FOMs? What does an application developer need to do when modifying the FOM, or switching to a different FOM?

3.1 The FOM-specific Code Generation Approach

The first approach we'll look at is the FOM-specific code generation approach. Several tools have been developed in the HLA community that can read a FOM and automatically generate FOM-specific middleware code. The API that is generated typically provides access to the current values of the attributes of local and remote objects, in the same form as the data representations in the FOM.

For example, if the FOM contains an attribute called ObjectMass, with data type double, the generated API might contain functions that look like this:
    void setObjectMass(double val);
    double getObjectMass();

But because the API so closely mirrors the layout of the FOM, when the FOM changes, the API needs to be regenerated. When this happens, each application that uses the generated API needs to be modified and recompiled to deal with the new middleware API.

The architecture of a system built on the code-generation approach appears in Figure 1.1. Changes in the FOM (and generated API) may be syntactic (when the

name or data type of an attribute changes) or semantic (when the meaning of the value of an attribute changes, e.g. a number used to represent pounds, and now represents kilograms).

The fact that information on the units to be used for each attribute is available in the FOM does not really help with code generation. The application itself needs to understand the semantics of the generated API and, if necessary, convert from its own local representation to the representation expected by the FOM.

3.2 FOM-configurable Fixed-API Middleware

An alternative to a FOM-specific middleware API of the type that is often produced by a code-generator, is a fixed[1], FOM-Independent API, supported by an architecture that allows you to plug in mappings to various FOMs without changing the API. This architecture is shown in Figure 1.2.

In this architecture, the middleware API used by the application remains the same regardless of how the data is represented in the FOM. This means that application code does not need to change when the FOM is modified, or when switching to a different FOM altogether. In order to switch to a different FOM, the middleware toolkit is "configured" for that FOM by changing the FOM Mapper layer (which is not seen by application code.)
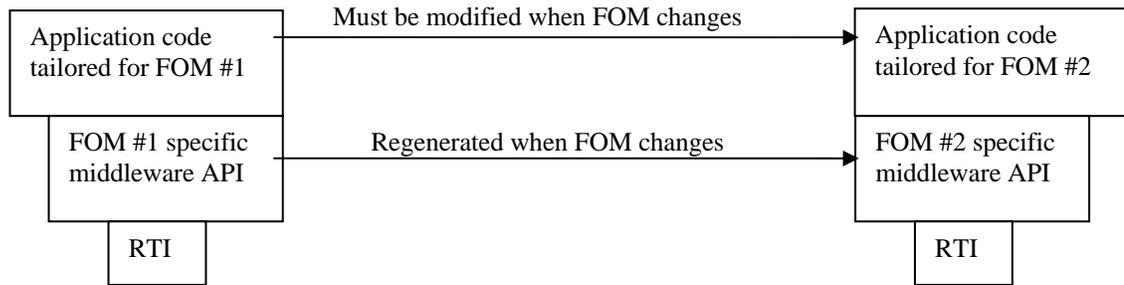
Several implementations of this approach (or variations) have been developed in the HLA community, including MAK's VR-Link toolkit and the DMSO-funded Agile FOM Framework, which is embedded in ModSAF.

There is some work involved in producing a FOM Mapper for a new FOM that you would like to support, but that work need only be done once, rather than n times for n applications. What's more, if the FOM Mapper functionality is placed in a shared library (DSO or DLL) that is used by the middleware, then one can switch among different FOM Mappers (and therefore different FOMs) without recompiling an application. This means that you can convert an application to be able to play with a new FOM even if source code for the application is not available.

Exactly what the middleware API looks like is fairly inconsequential. It might closely mirror one particular FOM that the API developer knows is likely to be used in many cases (which would make the job of developing the FOM Mapper for that particular FOM easier), or it might have been designed without any particular FOM in mind. At the least, the API developer must know what simulation domain the API is targeted for, and must have

---

[1] In a flexible FOM-configurable toolkit, like MAK's VR-Link, the top-level API is not truly fixed, but rather can be modified or extended to deal with different or new simulation concepts or domains. Of course, when new concepts are added to the API, application code must change if it wants to take advantage of the new functionality. The API is only fixed in the sense that it does not need to change when the FOM's data representation for a particular concept changes.

1.1) **The FOM-specific API Approach**

```
┌──────────────────────┐  Must be modified when FOM changes  ┌──────────────────────┐
│ Application code      │────────────────────────────────────▶│ Application code      │
│ tailored for FOM #1   │                                     │ tailored for FOM #2   │
├──────────────────────┤                                     ├──────────────────────┤
│ FOM #1 specific       │  Regenerated when FOM changes        │ FOM #2 specific       │
│ middleware API        │────────────────────────────────────▶│ middleware API        │
├──────────────┐        │                                     ├──────────────┐        │
│     RTI      │        │                                     │     RTI      │        │
└──────────────┘        │                                     └──────────────┘        │
```

1.2) **The FOM-configurable Fixed-API approach**

```
        ┌──────────────────────┐
        │  Application code     │◀──────  Does not need to change when FOM changes
        ├──────────────────────┤
        │ FOM Independent       │◀──────  Does not need to change when FOM changes
        │ Fixed Middleware API  │
        └──────────────────────┘
           ╱            ╲
    ┌──────────────┐  ┌──────────────┐
    │ FOM Mapper   │  │ FOM Mapper   │ }  Appropriate FOM Mapper swapped-in
    │ for FOM #1   │  │ FOM #2       │ }  with no change to Application code
    └──────────────┘  └──────────────┘
           ╲            ╱
          ┌──────────────┐
          │     RTI      │
          └──────────────┘
```
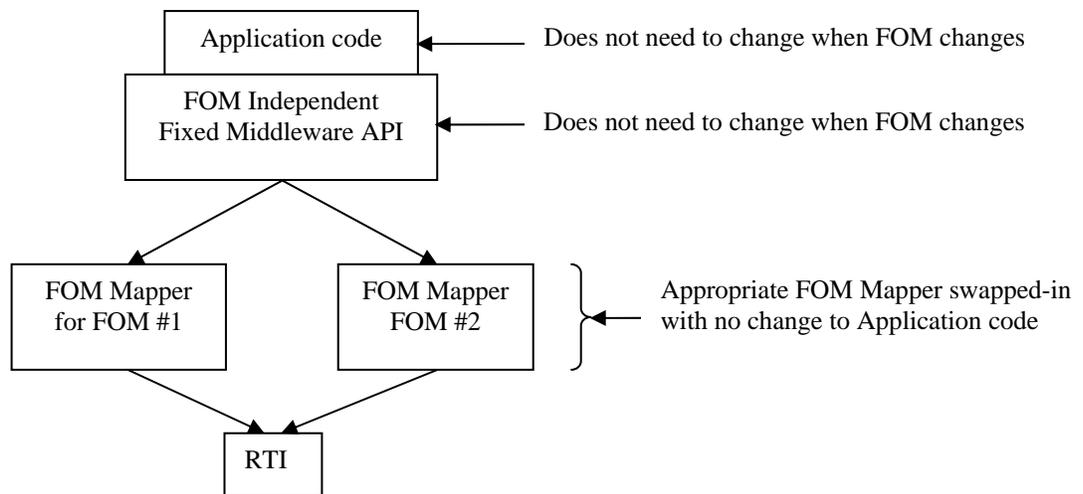
Figure 1: Comparing two middleware approaches. **1.1)** Using a FOM-specific middleware API, each application must be reworked when the FOM changes. **1.2)** Using a fixed-API, FOM-configurable toolkit, a new FOM Mapper is swapped in, without modifying application code.

some general sense of what objects and attributes applications will be dealing with. However, he does not need to know what the names or exact data types of classes and attributes will be in any particular FOM.

Even if the API developer does have a particular FOM in mind, it is often best if the API does not exactly match FOM representations, which are often chosen to minimize bandwidth rather than to facilitate an intuitive API.

One possible implementation of a FOM Mapper (the implementation used by MAK Technologies' VR-Link) is a table of encoding, checking, and decoding functions - one set of functions for each attribute of each class. When generating an outgoing attribute update, the middleware toolkit asks the FOM Mapper for a checking function that checks whether the update condition holds for a particular attribute, and an encoding function that converts the attribute from the FOM-independent API's representation to the current FOM's representation. When the middleware toolkit receives an incoming attribute update,

it asks the FOM Mapper for a decoding function that does the opposite conversion.

The toolkit is "configured" for use with a particular FOM by writing encoding, checking and decoding functions that are appropriate for the FOM, and registering those functions with the FOM Mapper.

4. Can a FOM Mapper be Automatically Configured?

In the introduction, we pointed out that the ultimate in flexibility would be if an application could automatically play with any arbitrary FOM, without a user having to do anything to the application.

We now know that the FOM-specific API code generation approach does not get us there, because application code must change when the FOM changes. But what about the FOM-configurable fixed-API approach? In this approach, only the FOM Mapper - and not application code - needs to change when switching

FOMs. So the question remains: Can we somehow automatically generate a FOM Mapper from a new FOM? The answer is unfortunately, no.

Mapping from FOM attributes to middleware accessor function calls requires an understanding of the semantics of the FOM, the semantics of the middleware API, and how they relate, which necessitates human intervention.

This could mean either writing FOM Mapper code from scratch, filling in the bodies of automatically generated function templates, encoding the mapping knowledge in a (rather complex) configuration file, or possibly even using a GUI to specify the relationships between FOM attributes and the middleware API. But in any case, it can't be done entirely automatically.

No tool can "know" that a FOM attribute name, such as Location, and a middleware accessor function name, such as setPosition both refer to the same concept (unless that bit of knowledge is hard-coded). Similarly, no tool can automatically "know" how to convert from one set of units to another set of units that it has never heard of before.

5. Conclusions

In this discussion, we have looked at two different middleware approaches: a FOM-specific API, which can be automatically produced by a code generator, and a

FOM-independent fixed API, that can be configured to support a particular FOM through an appropriate FOM Mapping plug-in.

completely FOM-independent, the APIs that they generate are FOM-specific. The advantage of this is that your application sees an API that very closely mirrors the FOM, which may be desired if this is the only FOM the application will be used with. However, this approach does not make your application FOM-agile. Application code must be reworked in order to switch to a different or modified FOM.

By contrast, a FOM-configurable fixed-API middleware toolkit does provide FOM-agility to your applications, solving the key problem we posed when we opened this discussion. Although changing to a new FOM requires generating a FOM Mapper for that FOM, the FOM-configurable fixed-API approach allows you to build your application once, and use it in multiple federations without recoding.

6. Author Biography

LEN GRANOWETTER is the lead software engineer in MAK Technologies' COTS products development group. He is the chief architect of MAK's VR-Link toolkit, and of the MAK Real-Time RTI.