A Helper Framework for Simplifying HLA Interfaces

Mr. Ryan Brunton  
The Johns Hopkins University Applied Physics Laboratory (JHU/APL)  
ryan.brunton@jhuapl.edu

Dr. Katherine L. Morse  
The Johns Hopkins University Applied Physics Laboratory (JHU/APL)  
katherine.morse@jhuapl.edu

ABSTRACT: The Environmental Biothreat Aerosol Detection System (EBADS) Federation is a federated simulation based on the High Level Architecture (HLA) for Modeling and Simulation. A federated simulation approach was selected to modularize models into separate simulations to enable replacement representation of EBADS critical technology elements (CTEs), which in turn allows independent development of CTE simulations. This creates the opportunity to reuse the best of breed existing simulations and potentially operational software. Because the EBADS Federation has nine simulation federates, several of which have detailed federation object model elements, and started with no existing federates, writing native HLA interfaces would have required considerable, repetitive coding. The Johns Hopkins University Applied Physics Laboratory federation engineering team developed a helper framework for simplifying encoding and decoding of data, time management, publication and subscription handling, and event logging. This framework, while created to support the development of the EBADS federation, was designed to quickly bootstrap any HLA federation development project with the necessary code to begin federate implementation without the need for low-level interaction with the HLA API to manage common federate functionality.
1 EBADS Federation Implementation

The EBADS Federation employs a custom HLA [1] helper framework developed by the Johns Hopkins University Applied Physics Laboratory (JHU/APL). This framework is designed to automate many of the cumbersome tasks associated with building HLA federates, including:

- Encoding and decoding data sent and received on the HLA Runtime Infrastructure (RTI)
- Managing subscription and publication of object attributes and interactions
- Time management and coordination of federate synchronization
- Logging simulation data

This is accomplished through a small set of helper classes coupled with some carefully chosen supporting frameworks and software applications.

The federation object model (FOM) object classes and interaction classes are illustrated in Figure 1 and Figure 2, respectively. Some classes are expanded to illustrate the level of detail but not all.

![Figure 1. FOM Object Classes](image-url)
The HLA standard was explicitly designed to allow implementation vendors to optimize the message wire encoding independent of the published application programming interface (API). As a result, reading and writing data using the API standard requires manually using an implementation-specific encoder and decoder for each object attribute and interaction parameter. These encoder and decoder implementations are specific to each primitive type defined in the specification and are created at the point of use in the code using static methods on implementation-specific factory classes. As the FOM for a simulation grows, this manual process can become extremely tedious and error prone and was the logical point for the framework to address first.

The JHU/APL framework for simplifying encoding and decoding of data consists of four parts:

1. A utility class with static encoder and decoder methods for all primitive types. Each method handles all of the HLA code necessary to generate the native encoder or decoder, perform the required operations, and return either a byte array ready to transmit or a Java primitive for use in the federate code. While these static methods can be called directly, there are also universal encode and decode methods, which infer the correct primitive type methods to call based on either metadata in the HLA API callback method or type information derived from the native Java object type.

2. A pair of abstract super classes, one for object classes and another for interactions, capable of parsing a message received from the RTI or encoding a message for transmission. The abstract classes utilize information pulled from

Figure 2: FOM Interaction Classes

2 Encoding and Decoding Data

The HLA standard was explicitly designed to allow implementation vendors to optimize the message wire encoding independent of the published application programming interface (API). As a result, reading and writing data using the API standard requires manually using an implementation-specific encoder and decoder for each object attribute and interaction parameter. These encoder and decoder implementations are specific to each primitive type defined in the specification and are created at the point of use in the code using static methods on implementation-specific factory classes. As the FOM for a simulation grows, this manual process can become extremely tedious and error prone and was the logical point for the framework to address first.

The JHU/APL framework for simplifying encoding and decoding of data consists of four parts:

1. A utility class with static encoder and decoder methods for all primitive types. Each method handles all of the HLA code necessary to generate the native encoder or decoder, perform the required operations, and return either a byte array ready to transmit or a Java primitive for use in the federate code. While these static methods can be called directly, there are also universal encode and decode methods, which infer the correct primitive type methods to call based on either metadata in the HLA API callback method or type information derived from the native Java object type.

2. A pair of abstract super classes, one for object classes and another for interactions, capable of parsing a message received from the RTI or encoding a message for transmission. The abstract classes utilize information pulled from
the RTI ambassador class (a component provided by the HLA specification used to interact with the federation and query for object model metadata) and Java introspection to determine what fields on their implementing subclasses map to corresponding elements of the FOM.

3. A concrete subclass extending one of the two abstract super classes. Each of these subclasses is a simple JavaBean [2], with fields named according to the corresponding elements in the FOM. While the JHU/APL team initially generated these manually for the EBADS federation, a utility was subsequently developed to automate this code generation process.

4. A pair of change tracking aspects, utilizing the AspectJ [3] framework, woven into all of the concrete subclasses described in part three at compile time, which allow for the HLA RTI encoding of only those object variables that have been explicitly locally modified.

The Unified Modeling Language (UML) [4] class diagram for these framework encoding and decoding support classes is provided in Figure 3.

![Figure 3. Framework Encoding and Decoding Classes](image)

As a result of using this part of the framework, data received as the result of a federate subscription can be decoded from the RTI and converted to or from a Java object into a byte array for transmission on the RTI with a single line of code (the `parse` method on either `ObjectClassWrapper` or `InteractionValuesWrapper` in Figure 3) instead of a several lines per object attribute or interaction parameter. Further, type mapping is handled automatically, reducing potential coding errors, and change tracking on local object class instances ensures transmission efficiency.

### 3 Auto-generation of Wrapper Implementations

As mentioned in the previous section, while the concrete wrapper implementation was initially coded manually by the JHU/APL team for the EBADS federation, the framework was later extended to allow these classes to be generated automatically. Since the majority of the functionality was already implemented in the abstract super classes, very little additional code was needed for the concrete object class or interaction wrappers. A simple utility class was created to parse the FOM file Extensible Markup Language (XML) [5]. For each object class, interaction, and enumeration, the parsed values
were passed to a template engine to generate the necessary Java code. To minimize code verbosity, the templates for the wrapper implementation Javabeans omitted accessor and mutator methods, adding these at compile time using the Lombok framework [6]. Class and enumeration names were generated automatically using the FOM names.

The wrapper code generated for the HLAObjectRoot.AnomalyAlgorithm object class in Figure 1 is shown below in Figure 4.

```
@Getter @Setter
public class AnomalyAlgorithmAttributes extends ObjectClassWrapper {
    private AnomalyDetectionStatusEnum anomalyDetected = AnomalyDetectionStatusEnum.Off;
    private double confidenceLevel = 1.0;
    private int alarmScore = 0;
    private HealthAndStatusEnum healthAndStatus = HealthAndStatusEnum.Healthy;

    @Override
    public String getFOMClassName() {
        return "HLAobjectRoot.AnomalyAlgorithm";
    }

    public AnomalyAlgorithmAttributes() {
        super();
    }
}
```

**Figure 4. Example Wrapper Code**

The @Getter and @Setter class annotations are the decorators imported from the Lombok framework, and will result in public getter and setter methods being generated in the class bytecode at compile time.

### 4 Object-Oriented Best Practices

Once the encoding and decoding framework was in place, the rest of the EBADS federation code was designed to maximize reuse and minimize boilerplate code. Common code for time management, joining and resigning from the federation, initializing factories, and registering synchronization points was implemented in the abstract base federate class included as part of the reusable framework. Simulation-specific code common to all federates was then implemented in an abstract EBADS federate from which each simulation federate implementation inherited. As a result, individual federate classes each contained only code necessary for their simulation function. Figure 5 illustrates the components of the federate architecture hierarchically, with each class to the left inheriting or implementing functionality from the class or interface to its right.

**Figure 5. EBADS Federate Architecture**

### 5 Managing Subscription and Publication

HLA subscription and publication consists of three major components:

1. Registering subscription and publication intent for each federate with the RTI
2. Handling callbacks from the RTI when subscribed data is received
3. Sending data that the federate publishes to the RTI
Normally, registration of subscription and publication intent requires a number of individual steps. The federate needs to retrieve the object or interaction class handle by name from the RTI ambassador. Then, for object classes, each attribute handle needs to be separately retrieved. Finally, the RTI ambassador is called again to register intent to subscribe or publish that data using the retrieved handles. As this is boilerplate code that is repeated identically for every object class or interaction a federate subscribes to or publishes, the JHU/APL team created helper methods that take as arguments a list of object classes or interactions as a string array, along with subscription and publication flags, eliminating the need to recreate this code for every subscription and publication for every federate. For object class attributes, the API method takes four arguments:

1. The fully-qualified name of the object class from the FOM
2. A string array containing the names of the attributes of interest
3. A boolean value indicating whether the federate will subscribe to these attributes
4. A boolean value indicating whether the federate will publish these attributes

Figure 6 illustrates the FSD federate publishing all the attributes of the FieldScreeningDevice object class.

```java
publishAndSubscribeAttributes("HLAobjectRoot.FieldScreeningDevice", new
String[] {"HealthAndStatus", "FSDResults", "FSDType", "FSDID",
"CollectorID"}, false, true);
```

Figure 6. Example Use of Publication / Subscription Helper

The methods for handling callbacks for subscriptions and sending updates for publications were also abstracted to a common superclass extended by every federate. Due to the previously described encoding and decoding framework, receiving data from the RTI requires only a single line of code to return a Java object populated with all the received data. Sending published data similarly benefits from the encoding framework, but has an added component. Using the Java AspectJ library, aspects were created that were added to the code at compile time, which added tracking data to each field of the Java objects representing object classes and interactions. This allowed the JHU/APL team to automate a federation engineering best practice for data transmission, encoding and sending only fields that were locally updated.

6 Time Management and Federate Synchronization

Time management and managing federation synchronization points can be error prone and, like many parts of the HLA API, require a fair amount of boilerplate code to implement for each federate. To avoid costly and error-prone code duplication across federates, helper methods were added to the common federate superclass to handle time management tasks while allowing for federate-specific lifecycle activities to be added to each federate implementation. Figure 7 illustrates the helper method for advancing to a specific time.

```java
protected void advanceToTime(double newTime) throws RTIexception {
    // request the advance
    isAdvancing = true;
    HLAfloat64Time time = timeFactory.makeTime(newTime);
    rtIambassador.timeAdvanceRequest(time);
    while(isAdvancing && !isShutdown){
        rtIambassador.evokeMultipleCallbacks( 0.001, 0.2 );
    }
}
```

Figure 7. Example Time Management Helper Method

7 Logger

Because the EBADS federation was primarily intended to support analysis of existing sensors and methods for aerosol bioterror detection, extensive logging was required so the state of the sensors, the aerosol dispersion, and the response processes could be examined by analysts throughout the simulated event lifecycle. While existing logging tools are provided by both RTI vendors and third parties, they proved limiting for the EBADS program. Some had proprietary log formats that could only be utilized by a corresponding tool from the same vendor. In the end, the encoding and decoding framework and the common federate superclass code previously developed provided a straightforward solution: a logger capable of being reused with the framework with any FOM.
The developed logger takes full advantage of the automatic encoding and decoding, as well as the dynamic schema capabilities of the MongoDB [7] NoSQL database, generating the necessary database schema and document classes at runtime. When the logger class is created by the logger federate, MongoDB document classes are created for each concrete wrapper class, themselves previously generated automatically from the FOM, using introspection. As part of the same code loop, the federate also subscribes to each object class and interaction for which a wrapper exists. As the simulation runs, each message from the RTI is automatically decoded using a matching wrapper class and saved as a MongoDB document after being augmented with the simulation ID and the message timestamp. The resulting log is completely tool agnostic and contains all simulation information transported across the RTI.

Figure 8 illustrates this process.

8 Conclusions and Recommendations

The EBADS Federation met its goal of providing a framework for modular integration of models of CTEs and processes. In addition to aligning with evaluation of operational requirements, key performance parameters, measures of success, and measures of effectiveness from the operational requirements document, the EBADS Federation could be expanded in the future to support operator training.

By basing the design on the open, international HLA standard, the federation engineering team ensured that future extensions won’t be hampered by vendor lock-in imposed by proprietary solutions. Considerable education, training, and technical resources exist for HLA as well as a labor pool of HLA developers. The code generation framework further simplifies the development of new simulation federates by encapsulating the repetitive details of some HLA APIs, which both reduces development and debugging time and increases the pool of developers who can come up to speed quickly on integrating models as simulation federates. The final architecture framework, conceptualized in Figure 9, reduces the amount of user federate code to only those elements specific to simulation functionality, allowing programs to prioritize federation functional design and results analysis over long development and debugging cycles.
9 Acknowledgements
This work was funded by the Department of Homeland Security (DHS) Countering Weapons of Mass Destruction (CWMD) Office.

10 References

11 Author Biographies
Dr. KATHERINE L. MORSE is a member of the Principal Professional Staff at the Johns Hopkins University Applied Physics Laboratory where she researches, designs, develops, and applies technologies for improving simulation engineering, implementation, and application. She received her B.S. in mathematics (1982), B.A. in Russian (1983), M.S. in computer science (1986) from the University of Arizona, and M.S. (1995) and Ph.D. (2000) in information & computer science from the University of California, Irvine. Dr. Morse has worked in computing for over 45 years, more than 25 of them contributing to open international standards. She is a Fellow of the IEEE.

Mr. RYAN BRUNTON is a Senior Staff Engineer at the Johns Hopkins University Applied Physics Laboratory. He received his B.S. in computer science (2001) from the University of California, San Diego and M.S. in Homeland Security from San Diego State University (2012). He has an extensive background in software design and implementation, computer security, open-source intelligence analysis, and HLA-based simulation design and development.