A Helper Framework for Simplifying HLA Interfaces

Ryan Brunton
Katherine L. Morse, PhD
Background and Motivation

• The Environmental Biothreat Aerosol Detection System (EBADS) Federation is a federated simulation based on the HLA.

• 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 JHU/APL 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.
A federation 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.
Utility Class for Primitive Handling

• 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.
Object and Interaction Class Wrappers

- 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.
Auto-Generation of Wrapper Implementations

• A simple utility class was created to parse the FOM XML.

• 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.

• Class and enumeration names were generated automatically using the FOM names.

• 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.

```java
@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.Heathy;

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

    public AnomalyAlgorithmAttributes() {
        super();
    }
}
```
Change Tracking Aspects

• A pair of change tracking aspects, utilizing the AspectJ 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.

```
©  ParameterUpdateTracking
  m  changeTracked()   void
  m  changeTrack(JoinPoint)   void

©  AttributeUpdateTracking
  m  changeTracked()   void
  m  changeTrack(JoinPoint)   void
```
EBADS Federate Architecture

• 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.

• This figure 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.
Publication / Subscription Helper

- Normally, registration of subscription and publication intent requires a number of individual steps, including
  - Retrieving object and interaction class handles
  - Retrieving object attribute handles
  - Registering intent to subscribe or publish data using the retrieved handles.

- The helper method API method for object class attributes takes as arguments a list of object classes or interactions as a string array, along with subscription and publication flags:
  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

- The code below 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);
```
Time Management and Federate Synchronization

• Time management and managing federation synchronization points can be error prone and require a fair amount of boilerplate code to implement for each federate.

• 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.

• The code below 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( 0.0001, 0.2 );
    }
}
```
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 NoSQL database, generating the necessary database schema and document classes at runtime.

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.
Framework Architecture

- The final architecture framework, conceptualized here, 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.
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.

• The code generation framework 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.
Questions?

This work was funded by the Department of Homeland Security (DHS) Countering Weapons of Mass Destruction (CWMD) Office.