Adapting Functional Mockup Units for HLA-compliant Distributed Simulation

Faruk Yılmaz, Umut Durak, Koray Taylan

Roketsan Missiles Inc.
Ankara, Turkey
[fyilmaz|udurak|ktaylan]@roketsan.com.tr

Halit Oğuztüzün

Middle East Technical University
Ankara, Turkey
oguztuzn@metu.edu.tr

Abstract

Conceptual design of systems requires aggregate level simulations of the designed system in its operational setting. By this way, performance of the system and its interactions with the other entities in its environment can be evaluated. The complex nature of these simulations often requires distributed execution. IEEE 1516 High Level Architecture (HLA) is a widely accepted standard architecture for distributed aggregate level simulations. Functional Mock-up Interface (FMI) is a recent standardization effort that leads to a tool independent systems simulation interface that enables model reuse and co-simulation. This paper aims to present a method for developing HLA-compliant federates using FMI. The method enables a Functional Mock-up Unit to join an HLA-compliant federation as a member.

Keywords: Functional Mockup Interface; High Level Architecture; Distributed Simulation

1 Introduction

Systems development process starts with conceptual design phase in which designers create concepts and conduct trade off analysis. Modeling and simulation have always been essential tools for conceptual design. Early stage systems modeling aims to identify the system requirements and its interactions with its operating environment. Effect based models, integrated in a large scale operational settings are used to evaluate the performance of the system concerning the accomplishment of its mission. Simulation of the mission space of a system requires modeling large number of entities and often simulating them in a distributed fashion. IEEE 1516 High Level Architecture (HLA) standard [1] [2] [3] is commonly used to integrate loosely coupled models of the entities in a mission space.

The Functional Mock-up Interface (FMI) is a newly developed, tool-independent model interface standard [4] [5]. Its main purpose is to model reuse between various modeling tools and environments throughout the systems development phases. A simulation component conforming to FMI is called a Functional Mock-up Unit (FMU), whose contents include a model description file, user defined libraries, source codes, model icons and documentation.

FMI and HLA has completely different behavior. While HLA supports to work at process level, the master of the FMU does not care about the topics such as entity transfer, shared resource management, time synchronization or ownership management [10].

On the other hand, there is a potential to reuse existing FMUs as federates in an HLA-compliant distributed simulation, i.e. federation. By this way, FMI will also serve as a model interface for distributed simulation entities in the concept of design phase. Here in this study, we introduce a mechanism to develop Functional Mockup Unit Federates (FMUFd) from FMUs.

1.1 Related Work

As model based development of engineering systems are getting more popular, connecting engineering models to the distributed simulation environments is also becoming an important issue of concern [6][7]. There have been some attempts for developing such tools and methodologies. Closely related to our work,

there exist two particular efforts providing a mechanism for connecting models to HLA environments.

MatlabHLA-Toolbox [8] and HLA Blockset [9] are available toolboxes to provide HLA communication feature to the Matlab. With these toolboxes, modelers can create a federate, join a federation and start publishing and subscribing entities and events. However, these solutions can only work in Matlab environment.

In [10], authors introduce an approach to run FMI Co-Simulation environment over HLA. They employ HLA RTI as a master to synchronize the simulation that is composed entirely of FMUs. They define an Object Class derived from the interface specifications of all participating federates and let each federate out of an FMU publish or subscribe its required attributes. In contrast, our approach enables participation of FMUs in a federation with non-FMU members as well.

2 Functional Mockup Interface

Functional Mockup Interface provides an interface specification for simulation components called Functional Mockup Units. FMI provides two standard interfaces, namely, FMI for Co-Simulation and FMI for Model Exchange [4] [5].

While FMI for Model Exchange specifies the interface for callers with explicit or implicit integrators, FMI for Co-Simulation specifies the interface for simulation runnables that possess solvers in them. As we can view HLA Federates as standalone simulation runnables, this effort is based on FMI-Co-Simulation interface for federate development. In this work, the first version of the standard is used as the baseline [5].

2.1 FMI for Co-simulation

As mentioned above, FMI for Co-Simulation is a standard interface for the model output containing its solver inside. Therefore, the user does not need to know which integration method is actually employed to solve the ordinary differential equations within the model.

For each of the FMU in a co-simulation environment, the communication capabilities are configured in a model specific XML file, namely *ModelDescription.xml* file. Communication with an FMU can only be realized in a discrete communication point, which is a sampling point or a synchronization point of the FMU [5].

2.1.1 Computational Flow

As show in Figure 1, FMU co-simulation computational flow has three main states, namely Instantiation and Initialization, Running and Termination.

Instantiation and Initialization

A new FMU instance is created and initiated to be ready to run. Memory allocations and initial value setting for the FMU parameters are performed in this phase.

Running

In this phase, FMU model is executed via calling *doStep()* method. Intuitively, before running a step, FMU input parameters are set by calling *FMUSetXXX(...)* and after the completion of this step the model output parameter are read by the master via calling *fmiGetXXX(...)*.

Termination

The model component is unloaded and the memory is cleaned up in this phase.

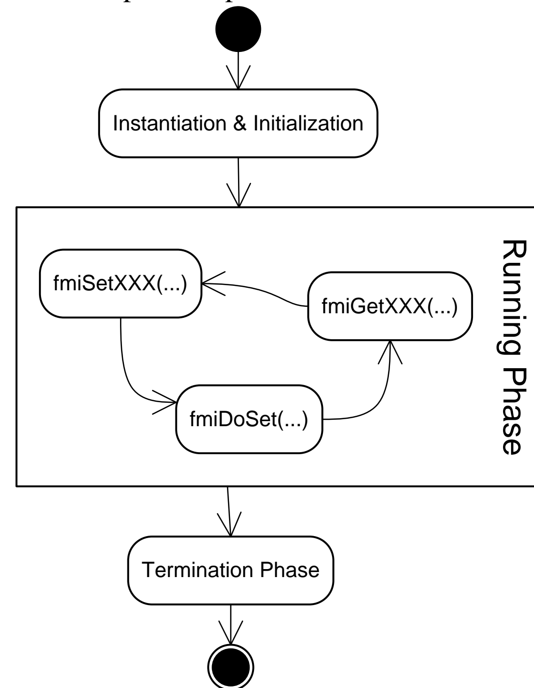

Figure 1 – FMU Co-Simulation Model Computational Flow

3 High Level Architecture

The High Level Architecture (HLA) is a common framework for distributed simulation systems. HLA promotes interoperability between heterogeneous simulations and supports the reuse of models in different contexts. HLA provides communicating data and synchronization actions between simulation members regardless of their computing platforms [1].

HLA combines simulations (federates) into a larger simulation (federation), where federates are

DOI 10.3384/ECP14096247

components and federations are component based applications. The HLA requires runtime infrastructure (RTI) software to support the operation of a federation execution. RTI provides a set of services and by using these services a federate can interact with the federation at runtime. How a federate can utilize these services is defined by the Federate Interface Specification [2].

Federation Object Model (FOM) is the HLA Federation Object Model that describes all of the object classes and interactions, attributes of object classes and parameters of interactions for the federation. Also, FOM establishes the information model contract which governs the simulation. Simulation Object Model (SOM), on the other hand, is the HLA Simulation Object Model that describes the object classes and interactions, attributes of object classes and parameters of interactions information which are exposed or consumed by a federate [2] .

3.1 HLA Services

HLA provides six groups of services to enable distributed simulation in an aggregate level [2]. Federation Management describes how to create, join, resign and manage federations, save and restore federation states. Declaration Management defines the publishing and subscribing to objects and attributes. Object Management service states how to register new instance of an object class or interaction, update the attributes, receive interactions, discover new instances and receive updates of attributes. Ownership Management defines acquisition of ownership of the registered objects. Time management describes how a federate can advance its logical time along with other federates and how to deliver the time-stamped events ensuring that a federate can never receive an event with a timestamp less than the federate's logical time. Data distribution management defines the production and consumption of data to bind the relevance of communication data among federates. As a result, RTI can recognize the irrelevant data and prevents its delivery to consumers.

3.2 HLA Object Model

HLA provides object classes and interactions as the object models, which are used to publish/subscribe the data over distributed simulation environment. Providing the data exchange between federates is one of the responsibilities of the RTI.

An **object class** can be derived from another object class. *HLAobjectRoot* is the base class of the all object classes. Each object class can contain one or more attributes. Derived classes also inherit base class attributes. Attributes are have data types. A federate will publish/subscribe only interested attributes of an object class; it does not have to deal with all the attributes in an object class.

An **interaction** can be derived from another interaction. *HLAinteractionRoot* is the base class of the all interactions. Each interaction contains one or many object parameters. Derived interactions takes base interaction parameters also. Parameters have data types. A federate should fill all the parameters of an interaction to publish it.

HLA provides six different **data types** where user can create variety of data structures by using those data types. The published/subscribed values are stored in these data structures. The details of data types are given below [3]:

- **Basic Datatype**: Basic data refers to a predefined set of data representations. Following data types should be defined by any OMT: *HLAinteger16BE, HLAinteger32BE, HLAinteger64BE, HLAfloat32BE, HLAfloat64BE, HLAoctetPairBE, HLAinteger16LE, HLAinteger32LE, HLAinteger64LE, HLAfloat32LE, HLAfloat64LE, HLAoctetPairLE, and HLAoctet*.
- **Simple Datatype**: The simple data type table refers to simple, scalar data items. Following data types should be defined by any OMT: *HLAASCIIchar*, *HLAunicodeChar*, and *HLAbyte*.
- **Enumerated Datatype**: The enumerated data type refers to data elements that can take on a finite discrete set of possible values. Following data type should be defined by any OMT: *HLAboolean*.
- **Array Datatype**: The array data type table refers to indexed homogenous collections of data types; these constructs are also known as arrays or sequences. Following data types should be defined by any OMT: *HLAASCIIstring*, *HLAunicodeString*, and *HLAopaqueData*.
- **Fixed Record Datatype**: The fixed record data type table refers to heterogeneous collections of types; these constructs are also known as records or structures. This allows users to build structures of data according to the needs of their federate or federation.
- **Variant Record Datatype:** The variant record data type table refers to discriminated unions of types; these constructs are also known as variant or choice records.

3.3 HLA Padding Rules

HLA requires that certain types of data start at a particular kind of location. Therefore, usually there is a requirement for extra bytes, namely padding bytes,

between data fields in a structure. To illustrate, consider a structure where the first field is a byte and second field is a double. Double must start at a position which is a multiple of 8. Therefore, seven bytes of padding is needed between byte field and double field for a proper structure.

The padding rules are used to determine exact positions of the fields of a data type, which constructs the data structure of an attribute.

These rules for constructed data types (arrays, fixed records, and variant records) as described below [3]:

Base Datatype

Each base type has a boundary value as provided in table 1. During the calculation of padding, this table is used to calculate structure boundary value.

Table 1 – Basic Datatype Boundary Values

Basic representation	Octet Boundary Value
HLAoctet	1
HLAoctetPairBE	2
HLAinteger16BE	2
HLAinteger32BE	4
HLAinteger64BE	8
HLAfloat32BE	4
HLAfloat64BE	8
HLAoctetPairLE	2
HLAinteger16LE	2
HLAinteger32LE	4
HLAinteger64LE	8
HLAfloat32LE	4
HLAfloat64LE	8

Simple Datatype

Same base data type padding rules also apply for simple datatype.

Enumerated Datatype

Same base data type padding rules also apply for enumerated datatype.

Fixed Record Datatype

The padding bytes are added to each field when necessary to ensure that the next field in the record is properly aligned. After a field the padding bytes can be calculated by using the following formula:

$$(Offset_i + Size_i + P_i) mod\, V_{i+1} = 0$$

where $Offset_i$ refers to the offset of the i'th field of the record as bytes,

$Size_i$ refers to the size of the i'th field of the record as bytes,

V_{i+1} is the octet boundary value of field $(i+1)$th of the record.

Variant Record Datatype

The *HLAvariantRecord* encoding shall consist of the discriminant followed by a field. This field is chosen by using the value of discriminant. The discriminant is placed at offset 0 of the record. The padding bytes P are calculated by using the following formula:

$$(Size + P) mod\, V = 0$$

where $Size$ refers to the size of the discriminant as bytes, and V refers to the maximum of the octet boundary values of the alternatives.

HLA Array Datatype with Fixed Cardinality

The padding bytes P_i between i'th and $(i+1)$th elements can be calculated by using following formula:

$$(Size_i + P_i) mod\, V = 0$$

where $Size_i$ is the size of the i'th element of the array in bytes,

V is the octet boundary value of the element type.

HLA Array Datatype with Variant Cardinality

The first 4 bytes are used to present the number of the elements in the array. These 4 bytes are encoded as *HLAinteger32BE*. The padding bytes can be added between the inform element and the first element of the sequence. The padding bytes can be found by using following formula:

$$(4 + P) mod\, V = 0$$

where V is the octet boundary value of the element type.

4 Functional Mockup Unit Federate Design

The FMI for Co-Simulation standard does not provide a specification for connecting FMUs to an HLA federation, hence, FMI Co-Simulation does not have an interface ready to utilize HLA services. Moreover, there is no convenient way to convert FMU scalar variables to HLA object class attributes, because FMI Co-Simulation only supports the following primitive types: *real*, *integer*, *string, Boolean* and *Enumeration*. On the other hand, HLA attributes can represent any data type structure, from basic data types to the complex data type structures. Since FMI Co-Simulation scalar variables can only map to HLA basic data types, a simulation environment using complex data types cannot be directly supported by FMI Co-Simulation.

DOI 10.3384/ECP14096247

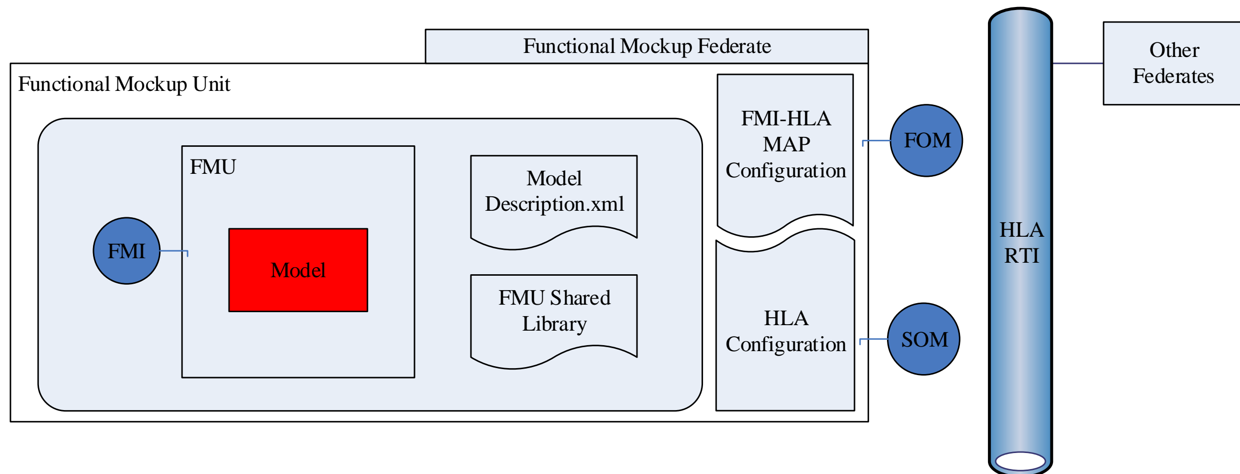

Figure 2 – Functional Mockup Unit Federate

Hence, there is a need for a wrapper that connects an FMU, in the context of FMI Co-Simulation, to the HLA distributed simulation environment. The work conducted handles the problem by designing a Functional Mockup Unit Federate (FMUFd). FMUFd has the following responsibilities:

- Instantiating, initializing, stepping and terminating of a FMU model.
- Providing the communication of distributed environment with services by using the HLA standard interface.
- Converting FMU model outputs to compatible HLA data types and sending them as HLA object updates.
- Receiving model inputs from HLA objects and converting them to compatible FMU types.

The top level structure of FMUFd that satisfies these requirements is depicted in Figure 2. The FMUFd is composed of the FMU model, FMI-HLA Map configurations and HLA connection configuration. FMI-HLA Map Configuration is used to inform FMUFd about HLA FMI relation. For each FMU, using this structure an FMUFd is needed to be configured. HLA connection configuration is related with the federation and FOM information of distributed simulation environment. By using these data, FMUFd runs with stepwise activities. As shown in Figure 3, these activities can be grouped into four main phases, namely, initialization, object discovery, stepping and termination.

In **initialization** phase, FMUFd loads and initializes the FMU and then connects the HLA federation as a federate with related HLA services and declare interested object classes for publishing and subscribing.

In **object reflection** phase, the subscribed object class instances are discovered and their values are reflected.

The **stepping** phase is the main phase of the simulation. In this phase, FMU input variables are reflected from related HLA objects, FMU runs one time step, and then, FMU output values are reflected to related HLA objects.

In **termination** phase, FMUFd terminates and unloads FMU, resigns from federation, frees allocated memory and finally stops.

The details of these steps with the process of connecting FMU to the HLA simulation environment will be described in the following sections. After that, the FMUFd capabilities in terms of the HLA services and FMUFd limitations will be mentioned briefly.

4.1 Loading an FMU

The loading of FMU takes two phases: In the first phase, the model description file is parsed, while the FMU is loaded and initialized in the second phase.

FMU Model description file provides the static information of all exposed variables and model related data. FMUFd uses model description file to identify scalar variables with data types and value reference, Globally Unique Identifier (GUID) and the model name. The scalar variables are used in data flow between FMUFd and FMU model. GUID is used for validating concrete coded FMU with model description file. Model name is used to load shared object and FMI functions. The dynamic link library has the same name as the model; shared object FMI functions should also take the model name as a prefix to their functions [5].

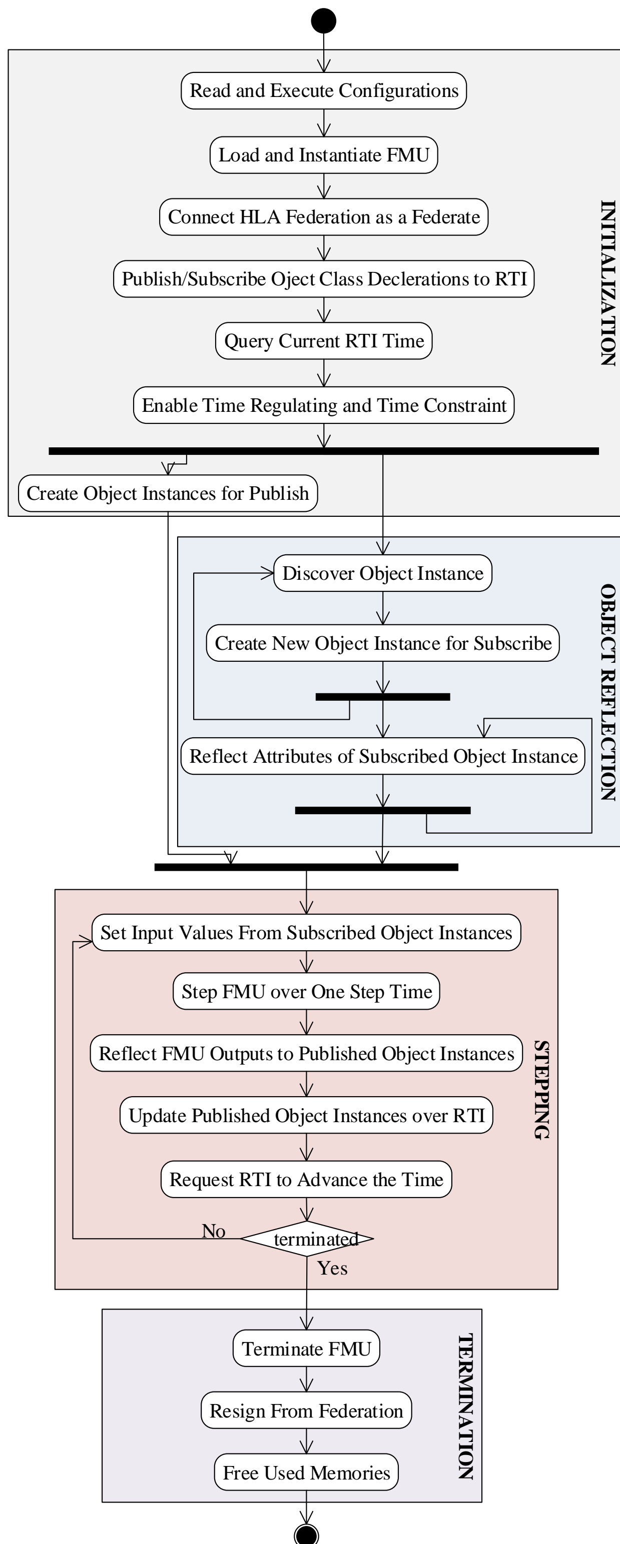

Figure 3– The FMUFd Activity Diagram

The FMU related operations are developed based on the FMU SDK [11]. By using these operations, FMUFd can load and use the FMU. The shared library, inside the FMU file should supply FMI Co-Simulation interface implementations. FMUFd loads those implementations automatically and then instantiates the model and gets the model instance. By then, FMU is ready to run steps over time.

4.2 FMU as an HLA Federate

This section describes how the FMUFd can join a federation execution as a member federate.

4.2.1 Connect to the HLA Federation

The FOM file contains all data exchange related information of HLA Federation, including object classes and object class attributes. By using this file, the FMUFd identifies the structure of each object class with attributes and data types.

The parsing process of a FOM takes two steps. First of all, the data types are parsed and stored in a map. For each type of the data, different parsing procedure is applied as each type has its special fields. For example, the size and endian information is set only for basic data types. Then, the object classes are parsed with their attributes and the data type of each attribute is retrieved from the map. If a class is derived, then its inherited attributes are obtained from its ancestors.

After parsing the FOM file, FMUFd tries to connect to the federation. The federation information is provided by the user through a configuration file. The FMUFd reads this file to get the federation name, path of FOM file and the name of its own federate. Then, FMUFd tries to create a federation if it has not been created yet. Finally, it joins the federation.

After joining the federation, FMUFd declares RTI which object classes with which attributes will be published and/or subscribed. This information is provided by the user with a SOM file. The FMUFd reads the file and identifies the published/subscribed objects and informs the RTI.

4.2.2 Create Object Instances

There are two scenarios for creating the object instances. At the beginning of the simulation, after declaring the object classes, the FMUFd creates the object class instances for publishing the FMU output parameters. The initial values of this object can be assigned by user with using the configuration files. Then, whenever an object class is discovered (new object instance is subscribed), the FMUFd creates an instance of the discovered object class.

Each object contains both object class metadata and attributes. Each attribute allocates the memory with the same size as its data type. While calculating the size of a data type the padding rules are used as described in section 3.3. Although there exists some

DOI 10.3384/ECP14096247

rules, still it may not be straightforward to find the exact size of the data type. For example, fixed record data type can contain another fixed record data type and a dynamic array data type. In this case, it is not possible to find exact size of the data type without filling the exact data. Therefore, for every update, the size of the data type should be recalculated. This recalculation may have a problem regarding the performance of an application. To address this issue, the FMUFd has been designed with two restrictions:

- The array data type with dynamic cardinality is not supported by FMUFd,
- The discriminant value of the variant record data type is explicitly defined in configuration file and cannot be changed in runtime.

With these restrictions, the FMUFd calculates the size of each attribute at the beginning of the simulation and uses this size throughout the simulation. As the data can contain different data types in it, the calculation may be performed through recursion. The basic data type is the only type with known size. Whenever the recursion reaches a basic data type, the padding rules are applied. The base case of the recursion could be the code segment given in Figure 4. The *currentOffset* value is passed into recursion which holds the previously calculated offset. After recursion is finished, *currentOffset* will hold the size of the root data type.

```
if(theDataType->type ==
ObjectClass::Attribute::DataType::BasicData)
{
  int mod = theDataType->size;
  int padding = (theDataType->size
                       - (currentOffset % mod))
                       % mod;
  theDataType ->offset = currentOffset + padding;
  currentOffset = newDataType->size
                       + theDataType->offset;
}
```

Figure 4 – The base condition code snapshot for calculating the padding bytes

4.2.3 Update/Reflect Object Class Attribute

A complex data type can contain both big endian and little endian data types in it, independent from application computer's endian type. Therefore, before updating the object class attribute, the attribute values should be encoded to the right type of endian. Likewise, after reflecting the attribute, the value should be decoded to the computer endian type. The FMUFd always keeps the data with the same encoding of computer. By doing that, it becomes easier to use the data in an application. Whenever an attribute is needed to be updated, the attribute is encoded first then update operation is called. Likewise, whenever an attribute is reflected, the value of that attribute is decoded first and kept in decoded form in memory.

The encode/decode operation is also executed with recursion. The basic data type is the only type with known endian type. Whenever the recursion reaches to the basic data type, the swapping operation is applied. The base case of the recursion could be the code segment given in Figure 5. The *returnValue* and *rowData* are the void* data type values, with the same size of attribute. If the recursion is used for updating the attribute operation than *rowData* refers to the current value of the attribute, otherwise, it refers to the reflected value of the attribute. The *returnValue* refers to the encoded (or decoded) value of the attribute.

```
if(dataType.type
== OjectClass::Attribute
::DataType::DataTypeType::BasicData)
{
  if(currentNodeEndianType
    != dataType.endianType )
  {
    T_UINT8* returnValueOffset
      = (((T_UINT8*) returnValue )
        + dataType.offset);
    T_UINT8* rowDataOffset
      = (((T_UINT8*) returnValue )
      + dataType.offset);

    switch(dataType.size)
    {
    case sizeof(T_UINT8):
      *returnValueOffset = *rowDataOffset;
      break;
    case sizeof(T_UINT16):
      *((T_UINT16 *) returnValueOffset)
= _byteswap_ushort(*(T_UINT16 *) rowDataOffset);
      break;
    case sizeof(T_UINT32):
      *((T_UINT16 *) returnValueOffset)
= _byteswap_ulong(*(T_UINT16 *) rowDataOffset);
      break;
    case sizeof(T_UINT64):
      *((T_UINT16 *) returnValueOffset)
= _byteswap_uint64(*(T_UINT16 *) rowDataOffset);
      break;
    }
  }
```

Figure 5 – The base condition code snapshot for encoding/decoding the attribute values.

4.3 Running the Federate

After introducing how HLA data is de-marshalled, the next step is mapping FMU scalar variables to HLA basic data types. This mapping is performed through user configuration files. These files inform the FMUFd about which data from HLA will be set to FMU and which data from FMU will be published to HLA.

After mapping between FMU scalar variables and the HLA attributes, the stepping function can be executed.

Before running a step of FMU, the FMUFd updates each input variable of the model. The input values are obtained from an instance of related object class. If there is no instance for related object class then the FMUFd will wait for the instance of that related object.

After running a step of FMU, the FMUFd updates related attributes of the HLA objects by retrieving the values from related FMU scalar variables. Therefore, an FMU output values can be mapped to different HLA objects, which are controlled by the FMUFd. After value updates are finished, the FMUFd will request the RTI to publish those attributes.

4.4 FMUFd Structure

The FMUFd is implemented as an application. Inspired from paper [14], the application is constructed with three base layers as Figure 6.

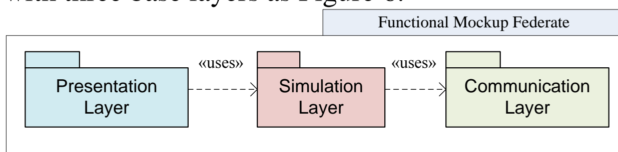

Figure 6 – the FMUFd Structure

4.4.1 Presentation Layer

The presentation layer is the user interface of the application. This layer provides presentation of application, input and interaction with the user as shown in Figure 7. The plot in the figure shows the change of some parameters of the missile and target over time. By using this layer, a user can load the necessary configurations to FMUFd and observe scalar variables' value changes in real time.

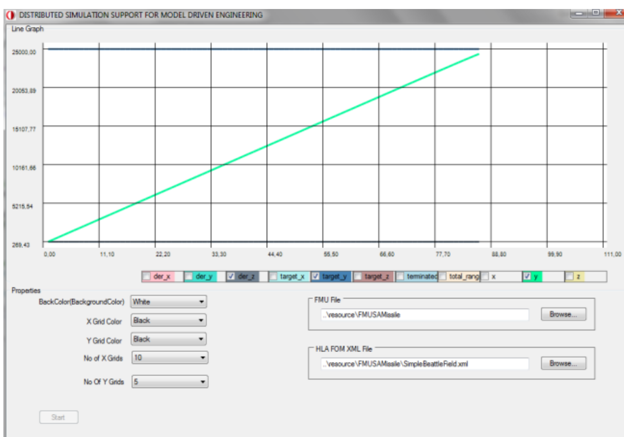

Figure 7 – The FMUFd screenshot while running the missile FMU

4.4.2 Simulation Layer

The simulation layer processes the application. It includes the computation of FMI simulation and federate specific HLA object classes. Its purpose is to run FMU and generate the federate behavior.
Simulation layer is responsible for running the simulation. This layer initializes the FMU, supplies necessary inputs for FMU from HLA class instances, runs the models and publishes the model outputs over HLA distributed environment.

One of the key features of the simulation layer is to create HLA object class structure dynamically. That is, without having the real structure, simulation layer can create a void data with the same size of the structure by using FOM xml file. Then simulation layer can edit this void data parts with the same position of any object class attribute fields.

4.4.3 Communication Layer

The communication layer deals with the RTI communication in order to access the object classes and interactions exchanged in the federation execution. RTI is the middleware that manages the federation execution and object exchange through a federation execution. In addition to data exchange, communication layer also supports time management service.

4.5 FMUFd Capabilities

In this section, FMUFd capabilities are described in terms of HLA interface services. Data distribution management and ownership management are not used in our current implementation.
Federation Management: If the federation has not been created before, the FMUFd creates the federation. Then it joins the federation. Similarly, after simulation is finished, the FMUFd resigns from the federation and if there is no other federate connected to the federation, it destroys the federation.
Declaration Management: FMUFd informs the RTI about publishing/subscribing object classes with attributes.
Object Management: Whenever new object class instance is discovered, FMUFd keeps the handle for this instance and allocates memory for it. Whenever a *reflectAttributeValues* event is raised by RTI, the FMUFd check whether the object instance is discovered before. If it is discovered, FMUFd reflects the attribute values to the allocated memories of the object instance, and ignores otherwise. Whenever a *removeObjectInstance* event is raised by RTI, the FMUFd checks if the object instance is discovered before. If it is discovered, FMUFd deletes the handle of instance and frees the related allocated memory.

DOI 10.3384/ECP14096247

FMUFd reflects the attribute values to the allocated memory of the object instance, ignores otherwise.
Time Management: FMUFd works as a time regulating and time constraining federate. As the nature of the time constraint, FMUFd ensures that the subscribed object model instance received reflection no less than the *currentRTITime.* Also, after each running step of the model, FMUFd requests to update the federate time.

4.6 Limitations

FMUFd still has some limitations and constraints. First of all, HLA interactions are not supported by FMUFd as there exists no corresponding logical concept in current FMI for Co-Simulation standard. Moreover, the array data type with dynamic cardinality is not supported by FMUFd. Finally, the discriminant value of the variant record data type is defined at the beginning of the simulation and FMUFd does not allow changing it at runtime.

5 Demonstration

To demonstrate the FMUFd usage, a simple distributed simulation environment is developed with MAK HLA RTI [12] implementation. For this application, the RPR2-D17 FOM file developed by SISO [13] is used as a FOM file. The used HLA object classes with its parent hierarchy are shown in Figure 8.

```
HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.Aircraft
HLAobjectRoot.BaseEntity.PhysicalEntity.Munition
```

Figure 8 – RPR2-D17 FOM classes used in the case study.

There are three nodes connected over an Ethernet network in this distributed simulation environment as shown in Figure 9. In the missile node, the missile co-simulation FMU (called *MissileFMUFd*) is connected to distributed simulation environment as the HLA federate by using FMUFd. Similar to missile PC, the aircraft co-simulation FMU is also connected to the simulation environment as a federate by using FMUFd (called *AircraftFMUFd*) in the target aircraft PC. The synthetic environment node is used to provide other entities in this operational setting, such as the missile launch platform, and to visualize the simulation in 2D and 3D. To this end, Presagis STAGE is used [15].

With two configuration files, one for FMU inputs and one for FMU outputs, user should supply information to the FMUFd about mapping between FMU scalar variables and HLA basic data types. Therefore, the entire hierarchy down to the basic data types should be explicitly defined for an object model.

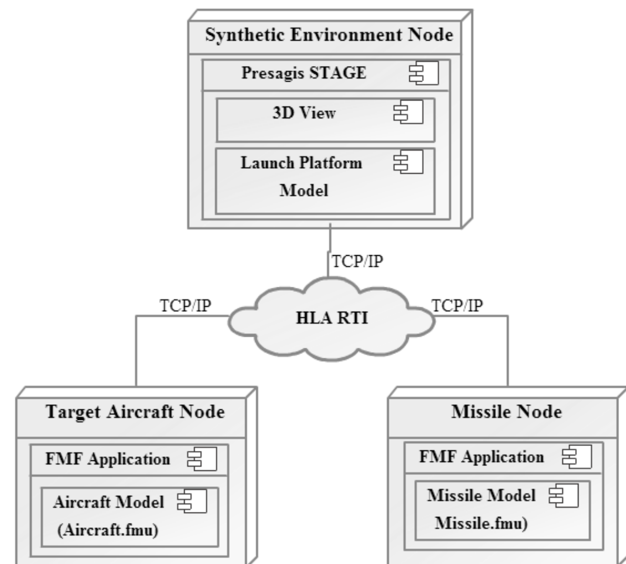

Figure 9 – The Deployment View Diagram of Simulation Environment

The example extract from a configuration file is provided in Figure 10. This example shows how the *Target_Ecef_X* scalar variable can be mapped with the *Aircraft* object's *Spatial* attribute's data type where data type goes down the hierarchy until it reaches the basic data type *HLAfloat64BE*. This mapping is specified for other scalar variables as well.

```
Target_Ecef_X =  Aicraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives
:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]
:SpatialFPStruct:WorldLocation:WorldLocationStruct:X
:HLAfloat64BEmetersperfectalways:HLAfloat64BE
```

Figure 10 – The example mapping between FMU scalar variables and HLA object class attribute data types.

With these configurations, 1500 simulation runs have been executed and performance figures for framework overhead have been measured. The median time for updating FMU parameters from HLA objects for *MissileFMUFd* is 254 microseconds. Likewise, the median time for updating HLA attributes from FMU parameters for *MissileFMUFd* is 356 microseconds. Measurement were taken on a computer with Intel Xeon 2.66GHz processor, 4GB DDR3 RAM and Windows 7 Pro 64bit operation system.

6 Conclusion

The FMI is an emerging standard for co-simulation and model exchange in Model Based Integration community. Also, HLA is a well-accepted standard for the distributed simulation. FMUFd supplies a solution for participation of FMUs that implement FMI Co-Simulation interface in an HLA Federation. Thus, a system model can be simulated as a part of an aggregate simulation of its operational setting. Moreover, this promotes a high level of reusability of system models supporting FMI.

As an alternative approach, the wrapping process may generate an FMU HLA wrapper code automatically by reading the FMU specification and generating the wrapper that knows how to translate just the specific FMU join HLA. This approach, on the other hand, is both FMU and federation specific and requires a recompile for each FMU. In our case, we aimed at recompilation free integration of FMUs to any federation via configuration files.

Finally, FMUFd is currently released in Roketsan Inc. as an in-house developed simulation infrastructure and being employed in some system development projects.

Acknowledgments

This work is supported by Turkish Ministry of National Defense, Undersecretariat for Defense Industries [Project Name: MOKA].

References

[1] HLA Working Group of IEEE: IEEE Std 1516-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules. New York, USA, 2000.

[2] HLA Working Group of IEEE: IEEE Std 1516.1-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification. New York, USA, 2000.

[3] HLA Working Group of IEEE: IEEE Std 1516.2-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification. New York, USA, 2000.

[4] MODELISAR Consortium: Functional Mock-up Interface for Model Exchange Version 1.0, www.functional-mockupinterface.org, January, 2010

[5] MODELISAR Consortium: Functional Mock-up Interface for Co-Simulation Version 1.0, www.functional-mockupinterface.org, October, 2010

[6] Stenzel, C.: Distributed Simulation in Technical Applications, X International PhD Workshop, OWD 2008, Conference Archives PETiS, Vol. 25, 2008, Gliwice, Poland, 513-518

[7] Lasnier, G., Cardoso, J., Siron, P., Pagetti, C. and Derler, P.: Distributed Simulation of Heterogeneous and Real-time Systems. (2013) In: 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications - IEEE/ACM DS-RT 2013, 30 October 2013 - 01 November 2013 (Delft, Netherlands).

[8] Stenzel, C., Pawletta, S.: CERTI - Bindings to Matlab and Fortran, University of Wismar, http://www.mb.hs-wismar.de/~stenzel/software/MatlabHLA.html Germany, 2008 (Last Accessed 20/11/2013)

[9] HLA Toolbox ™ The MATLAB® interface to HLA, http://www.forwardsim.com

[10] Awais, M. U., Palensky, P., Elsheikh, A., Widl, E. and Matthias, S.: The High Level Architecture RTI as a master to the Functional Mock-up Interface components. Vienna, AUSTRIA, 2013

[11] QTronic Company Developer Team: FMU SDK version 1.0.2, http://www.qtronic.de/en/fmusdk.html, 2010

[12] MÄK Technologies: MAK RTI User's Guide, 2013

[13] Shanks, G.: Real-time Platform Reference Federation Object Model (RPR FOM) Version 2.0D17, Simulation Interoperability Standards Organization, 2003

[14] Topçu, O. and Oğuztüzün, H.: Layered simulation architecture: A practical approach", Simulation Modelling Practice and Theory (SIMPAT) Journal, vol. 32, March 2013, pp. 1-14.

[15] Stage Sales Team: Stage, A Complete Simulation Development Environment, http://www.presagis.com, Montreal, CANADA, 2013