FUNCTIONAL MOCKUP UNIT ADAPTATION FOR HLA-COMPLIANT DISTRIBUTED SIMULATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FARUK YILMAZ

IN PARTIAL FULFILLMENT OF REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2014

Approval of the thesis:

FUNCTIONAL MOCKUP UNIT ADAPTATION FOR HLA-COMPLIANT DISTRIBUTED SIMULATION

submitted by **FARUK YILMAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering Dept., METU** _____

Dr. Umut Durak
Co-supervisor**, Game Technologies, METU** _____

Examining Committee Members:

Prof. Dr. Veysi İşler
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Assist. Prof. Dr. Selim Temizer
Computer Engineering Dept., METU _____

Dr. Attila Özgit
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _____

Date: 17.01.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Faruk Yılmaz

Signature :

ABSTRACT

FUNCTIONAL MOCKUP UNIT ADAPTATION FOR HLA-COMPLIANT DISTRIBUTED SIMULATION

Yılmaz, Faruk

M. Sc., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

Co-Advisor: Dr. Umut Durak

January 2014, 76 Pages

Conceptual design of systems require aggregate level simulations of the designed system in its operational setting. Thus, performance of the system and its interactions with the other entities in its environment can be evaluated. The complex and heterogeneous 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 enabling model reuse and co-simulation. This thesis aims to present a method for adapting FMI-compliant units to HLA. The presented 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

ÖZ

HLA UYUMLU DAĞITIK SİMÜLASYON ORTAMI İÇİN İŞLEVSEL MODEL ARAYÜZÜNÜN UYARLANMASI

Yılmaz, Faruk

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün

Ortak Tez Yöneticisi: Dr. Umut Durak

Ocak 2014, 76 Sayfa

Sistemlerin kavramsal tasarımı işlevsel ortamda tasarlanan sistemin birleştirilmiş düzey benzetimlerini gerektirir. Bu şekilde, sistemlerin performansları ve çevresi ile olan etkileşimleri değerlendirilebilir. Karmaşık doğası gereği bu benzetimler genellikle dağıtık simülasyon ortamlarına ihtiyaç duyar. IEEE 1516 Yüksek Seviye Mimari (HLA) dağıtık simülasyonlar için geliştirilmiş ve geniş ölçüde kabul edilmiş standart mimaridir. İşlevsel Model Arayüzü (FMI) yakın zamanda geliştirilmiş model yeniden kullanılabilirliği ve araç bağımsız sistem simülasyon ara yüzü sunan bir standart çabadır. Bu tez FMI kullanan HLA uyumlu bir federasyon geliştirme yöntemini sunmayı hedeflemektedir. Bu yöntem bir İşlevsel Model Birimi (FMU)'nin HLA uyumlu bir federasyon ortamına bir üye olarak katılmasına imkân sağlar.

Anahtar Kelimeler: İşlevsel Model Arayüzü; Yüksek Seviye Mimari; Dağıtık Simülasyon

To My Family

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Halit Oğuztüzün for his guidance, advice, criticism, encouragements and insight throughout this research.

I also wish to thank a lot to my co-advisor Dr. Umut Durak for all the valuable knowledge, technical support, academic assistance, innovative ideas and making us feel that he is always there ready for help in case we needed.

I also wish to thank a lot to my leader Koray Taylan for all the valuable efforts to ease the procedural processes, to find financial contribution, to provide moral support, to enhance the quality of this thesis work.

I also would like to thank a lot to Assoc. Prof. Dr. Ece Schmidt for all the valuables comments and criticism for both thesis and demonstration work.

I also wish to thank a lot to Gökhan Bircan and Koray Küçük for all their valuable efforts on model and simulation environment development.

I would like to thank to gratefully acknowledge Memduha Aslan and Başak Akgün for their comments and helps on technical writing for this research.

I would like to thank to Turkish Ministry of National Defense, Undersecretariat for Defense Industries which gave the team financial and moral support [Project Name: MOKA].

At last, but the most, my special thanks are due to my wife Tuba Yılmaz for her endless patience.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF LISTINGS

LISTINGS

LIST OF ABBREVIATIONS

CENG	Computer Engineering
DDS	Data Distribution Service
DIS	Distributed Interactive Simulation
DoD	Department of Defense
EXCIMS	Executive Council for Modeling and Simulation
FEDEP	Federation Execution Process
FMI	Functional Mockup Interface
FMU	Functional Mockup Unit
FMUFd	Functional Mockup Unit Federate
FOM	Federation Object Model
HIL	Hardware in the Loop
HLA	High Level Architecture
ITEA	Information Technology for European Advancement
MAP	Modelica Association Project
MDE	Model-Driven Engineering
METU	Middle East Technical University
MOM	Management Object Model
OMT	Object Management Template
PLM	Product Lifecycle Management

SISO	Simulation Interoperability Standards Organization
SOM	Simulation Object Model
TRADT	Time Representation Abstract Data Type

CHAPTER 1

INTRODUCTION

This chapter introduces the motivation and scope of the study, summarizes the related works and further outlines the organization of the thesis.

1.1 Aim and Scope of the Study

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 their simulation often requires a distributed environment. To integrate the models of the individual simulations of entities on mission space IEEE 1516 High Level Architecture (HLA) standard [1] [2] [3] is commonly used.

The Functional Mock-up Interface (FMI) is a newly developed, tool-independent model interface standard [4] [5]. Its main purpose is 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.

There is a potential for utilizing this simulation tool and environment independent standard for developing HLA federates. By this way, FMI can also serve as a model interface for distributed simulation entities in the concept of design phase. Here in this study, we introduce a method to develop Functional Mockup Unit Federates (FMUFd) from FMUs. Thus, we will enable to simulate system model as a part of an aggregate simulation of its operational settings. Moreover, this study promotes a high level of reusability of system models supporting FMI.

There are some other works carried out to join models in a distributed simulation. MatlabHLA-Toolbox [8] and HLA Blockset [9] are two different toolboxes offering HLA communication feature to the Matlab. In [10], the author introduces a concept for using HLA RTI as a FMI Co-simulation master. This concept defines how to use HLA RTI services as a master. In [12], the author states a simulation environment for developing a missile where a hardware-in-the-loop (HIL) simulation system, based on high Level Architecture (HLA) and Modelica language, are used. In [13], the author declares the SPRINT project which is a corporation of six European companies. The goal of the project is to simplify collaborative systems engineering across tools and platforms and to validate the systems including physical devices based on distributed real time simulation using HLA and FMI.

1.2 Organization of the Thesis

The preceding sections of this chapter introduce the motivation and scope of the study, present the summary of the thesis application and the related works. The remaining chapters are branched as follows:

- Chapter 2 provides related literature and background information required for easy understanding of the subsequent chapters. The HLA and FMI (especially FMI for Co-simulation concept) is briefly explained in this section. Additionally, programming for HLA and FMI is exemplified in this section.

- Chapter 3 explains the related works in details and discusses the works with our thesis work.

- Chapter 4 explains the details of FMU federate architecture. This section, firstly, presents details of mapping an FMU into the FMUFd architecture. Then, it explains how an FMUFd can connect to a federation as a federate. Further, it mentions mapping between FMI and HLA object classes. Moreover, this section informs about the implementation details of the FMUFd. Furthermore, this section evaluates the capabilities of FMUFd in terms of HLA services. Finally, this section lists the limitations of FMUFd.

- Chapter 5 explains a demonstration for the usage of the FMUFd in a simulation environment. In the demonstration, there are three nodes together with two FMUFd applications and one third party application.
- Finally, Chapter 6 discusses the accomplishments and draws conclusions.

CHAPTER 2

BACKGROUND

2.1 High Level Architecture (HLA)

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

HLA combines simulations (federates) into a larger simulation (federation), where federates are 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 during runtime. How a federate can reach the services are defined by the Federate Interface Specification [1].

2.1.1 History

Before 1988 there were so many military simulations; however, they provided a limited scope for simulation and they had little interoperability. Interest in distributed simulation, in 1988 DARPA's SIMNET (Simulator Network) Program was adverted. In 1991, development of Distributed interactive simulation (DIS) was initiated in a series of "Workshops DIS" in the Interactive Network simulation training symposium, organized by the University of Central Florida Institute for Simulation and Training (IST). After 1994, it was recognized that DIS did not provide a solution for the requirement of distributed simulation, like time management. In 1995, the Executive Council for Modeling and Simulation (EXCIMS) decided to establish an Architecture Management Group (AMG) to develop the High Level Architecture (HLA) and on 06 September 1996, the baseline HLA was developed [14].

The first complete version of the standard, published in 1998 was known as HLA 1.3. After the year 2000, the HLA specifications are moved as IEEE standards with IEEE 1516 series. These series of standards are listed below [14]:

- IEEE 1516–2000 – Standard for Modeling and Simulation High Level Architecture – Framework and Rules
- IEEE 1516.1–2000 – Standard for Modeling and Simulation High Level Architecture – Federate Interface Specification
- IEEE 1516.2-2000 – Standard for Modeling and Simulation High Level Architecture – Object Model Template (OMT) Specification
- IEEE 1516–2010 – Standard for Modeling and Simulation High Level Architecture – Framework and Rules
- IEEE 1516.1–2010 – Standard for Modeling and Simulation High Level Architecture – Federate Interface Specification
- IEEE 1516.2-2010 – Standard for Modeling and Simulation High Level Architecture – Object Model Template (OMT) Specification
- IEEE 1516.3-2003 – Recommended Practice for High Level Architecture Federation Development and Execution Process (FEDEP)
- IEEE 1516.4-2007 – Recommended Practice for Verification, Validation, and Accreditation of a Federation an Overlay to the High Level Architecture Federation Development and Execution Process

2.1.2 Definitions

Object Model Template (OMT)

OMT is the Object Model Template that provides a mechanism for specifying data exchange within a federation. OMT defines the format and syntax for HLA object models. It also provides a mechanism for describing the capabilities of federate, like objects and interactions managed by a federate or visible outside the federate. OMT

facilitates interoperability among simulations and enhances reuse of simulation components. OMT defines the Federation Object Model (FOM), Simulation Object Model (SOM) and Management Object Model (MOM) [2].

Federation Object Model (FOM)

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 [2] .

Simulation Object Model (SOM)

SOM 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].

Management Object Model (MOM)

MOM is the HLA Management Object Model that provides a predefined set of information elements to be included in the FOM related with federation management [2].

Run-Time Infrastructure (RTI)

RTI is the Run-Time Infrastructure that is a software layer providing common services to the federates for synchronization and data exchange. RTI specifications define the interfaces that federates should use to obtain services and to interact with other federates in a federation [1].

2.1.3 HLA Rules

These rules are extracted from IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules [1]. HLA rules define the behavior and capabilities of federates and federations. There are a total of ten rules; five rules for federation and five rules for federates.

Federation Rules

- "Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT)."
- "In a federation, all representation of objects in the FOM shall be in federates, not in the run-time infrastructure (RTI)."
- "During a federation execution, all exchange of FOM data among federates shall occur via the RTI."
- "During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification."
- "During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time."

Federate Rules

- "Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT)."
- "Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM interactions externally, as specified in their SOM."
- "Federates shall be able to transfer and/or accept the ownership of an attribute dynamically during a federation execution, as specified in their SOM."
- "Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM."
- "Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation."

2.1.4 HLA Services

HLA provides six groups of services to enable distributed simulation in an aggregate level. Those are summarized below [1].

Federation Management

Federation Management service defines how a federate can connect to the RTI. It also describes how to create, join, resign and manage federations, save and restore federation states.

Declaration Management

Declaration Management service defines the publishing and subscription of objects and attributes. The object class and object instance declarations are made inside FOM.

Object Management

Object Management service defines how to register new instance of object class or interaction, update the attributes, receive interactions, discover new instances and receive updates of attributes.

Ownership Management

Object management service defines acquisition of ownership of the registered object. This service allows updating an instance of object models with different federates.

Time Management

Time management service defines how a federate can advance its logical time with other federates and how to deliver the time-stamped events ensuring that a federate can never receive an event with logical times less than federates logical time. This service also affects other services like object/interaction updates and federate saves.

Data distribution management

Data distribution management defines the production and consumption of data for binding the relevance of communication data among federates. As a result, RTI can recognize the irrelevant data and prevent its delivery to consumers.

2.1.5 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 exchanges among federates are 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 many attributes. Derived classes also inherit base class attributes. Attributes 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 take 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 a 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, 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.

2.2 HLA Programming in C++

HLA can be programmed in many different programming languages, like C++ and Java. This section will briefly inform about the programming HLA federate by using C++.

Some parts of the information, which will be represented in this section, are extracted from MAK RTI Users Guide [16].

2.2.1 Interface Classes

There are two main C++ interface classes to provide communication between HLA RTI and a federate, namely *FederateAmbassador* and *RTIAmbassador*. The *FederateAmbassador* creates an interface through which the RTI can callback the events to the federate. The MAK RTI provides a base class, namely *NULLFederateAmbassador*, which has empty implementations of all the callback

methods of *FederateAmbassador*. A federate developer can simply drive a class from *NULLFederateAmbassador* and overrides the interested callbacks. By using *RTIAmbassador*, a federate can call the RTI methods. A developer can get derived class of *RTIAmbassador* by using *RTIambassadorFactory* class. All communication between the federate and the RTI must proceed through the *FederateAmbassador* and the *RTIAmbassador*.

2.2.2 Initialization

The initialization process takes two steps: creating a federation (if it doesn't exist) and joining to the federation.

Create a Federation Execution

The Create Federation Execution service takes a federation name and a FOM file path as the arguments. The name of the federation will be used as key to federates while joining the federation. The only way of controlling the existence of a federation is using a try-catch block and checking whether RTI throws *FederationExecutionAlreadyExists* exception.

```
RTIambassadorFactory* rtiAmbFactory = new RTIambassadorFactory();
std::auto_ptr < RTIambassador > rtiAmbAP =
    rtiAmbFactory->createRTIambassador(args);
delete rtiAmbFactory;
RTIambassador* rtiAmbassador = rtiAmbAP.release();

std::wstring federationName(L"The Name of the Federation");
std::wstring federationFile(L"FOM file path");
try
{
    rtiAmbassador->createFederationExecution(federationName, federationFile);
}
catch(FederationExecutionAlreadyExists& ex)
{
    // Federation already exist
}
catch(rti1516::Exception& ex)
{
    // error
}
```

Listing 1 – Code snapshot for creating a HLA federation

Join the Federation Execution

Every federate should connect to a federation to be a part of the simulation. *joinFederationExecution* service takes three arguments; the first parameter is a label for defining federate, the second parameter is the name of federation, the third parameter is a class reference derived from the *FederateAmbassador* interface.

```
std::wstring federationType(L"The Name of the Federation");
MyFederateAmbassador federateAmbassador();
FederateHandle theFederateHandle;
try
{
    theFederateHandle =
        rtiAmbassador->joinFederationExecution(federateType, federationName, fedAmb);
}
catch(rti1516::Exception& ex)
{
    // error
}
```

Listing 2 – Code snapshot for joining a HLA federation as a federate

2.2.3 Declaration of Objects

Before using objects or interactions, a federate informs the RTI about the interested objects with attributes or interactions with parameters. RTI has a handle for every object class, attributes interaction and parameter. While declaring the objects, those handles are used to inform RTI.

The declaration of object process takes four steps: Publishing the object classes, subscribing the object classes, publishing interactions and subscribing the interactions.

Publish the Object Class

The federate gives a list of attribute handles to the RTI. This list means that the federate is only interested with the listed attributes to publish even object class declaration can have more attributes.

```
ObjectClassHandle theClassHandle;
ObjectInstanceHandle theObjectHandle;
AttributeHandle attributeHandle1;
AttributeHandle attributeHandle2;
theClassHandle = rtiAmbassador->getObjectClassHandle(L"object name");
attributeHandle1 =
    rtiAmbassador->getAttributeHandle(theClassHandle, L"attribute1 name");
attributeHandle2 =
    rtiAmbassador->getAttributeHandle(theClassHandle, L"attribute2 name");
AttributeHandleSet hSet;
hSet[L"attribute1 name"] = attributeHandle1;
hSet[L"attribute2 name"] = attributeHandle2;
rtiAmbassador->publishObjectClassAttributes(theClassHandle, hSet);
```

Listing 3 – Code snapshot for declaring HLA objects to RTI to list the attributes to be published

Subscribe the Object Class

The federate gives a list of attribute handles to the RTI. This list means that the federate is only interested with the listed attributes to subscribe even object class declaration could have more attributes.

```
ObjectClassHandle theClassHandle;
ObjectInstanceHandle theObjectHandle;
AttributeHandle attributeHandle1;
AttributeHandle attributeHandle2;
theClassHandle = rtiAmbassador->getObjectClassHandle(L"object name");
attributeHandle1 =
    rtiAmbassador->getAttributeHandle(theClassHandle, L"attribute1 name");
attributeHandle2 =
    rtiAmbassador->getAttributeHandle(theClassHandle, L"attribute2 name");
AttributeHandleSet hSet;
hSet[L"attribute1 name"] = attributeHandle1;
hSet[L"attribute2 name"] = attributeHandle2;
rtiAmbassador->subscribeObjectClassAttributes(theClassHandle, hSet);
```

Listing 4 – Code snapshot for declaring HLA objects to RTI to list the attributes to be subscribed

Publish the Interaction

The federate gives the handle of the interaction class to RTI to declare which interactions can be published by this federate.

```
InteractionClassHandle theInterClassHandle;
theInterClassHandle =
    rtiAmbassador->getInteractionClassHandle(L"interaction class name");
rtiAmbassador->publishInteractionClass(theInterClassHandle);
```

Listing 5 – Code snapshot for declaring HLA interactions to RTI, by then the federate can publish this kind of interaction later.

Subscribe the Interaction

The federate gives the handle of the interaction class to RTI to declare which interactions can be subscribed by this federate.

```
InteractionClassHandle theInterClassHandle;
theInterClassHandle =
    rtiAmbassador->getInteractionClassHandle(L"interaction class name");
rtiAmbassador->subscribeInteractionClass(theInterClassHandle);
```

Listing 6 – Code snapshot for declaring HLA interactions to RTI, by then the federate can subscribe this kind of interaction later.

2.2.4 Information Exchange

Information exchange contains following steps: update/reflect attribute values, send/receive interaction and time management (time advance request and time advance grant). The reflect attributes values receive interaction and time advance grant methods are obtained from *FederateAmbassador* callback methods.

Update Attribute Values

Any attribute value can be updated individually from the federate. The attributes that will be updated are listed with their values into *AttributeHandleValueMap*.

```
AttributeHandleValueMap attrValues;
void * dataPointer1; // fill data with some value
VariableLengthData variableLengthData1(dataPointer1, data1Size);
void * dataPointer2; // fill data with some value
VariableLengthData variableLengthData2(dataPointer2, data2Size);
attrValues[parameter1Name] = variableLengthData1;
attrValues[parameter1Name] = variableLengthData2;
std::string tag("Any String");
rtiAmbassador->updateAttributeValues(
    theObjectHandle,
    attrValues,
    VariableLengthData(tag.c_str(), tag.size()+1));
```

Listing 7 – Code snapshot for updating attribute values over HLA federation

Send Interactions

The parameters that will be sent are listed with their values into *ParameterHandleValueMap*. The parameter list of the interaction should be filled completely.

```
ParameterHandleValueMap  paramValues;
void * dataPointer1; // fill data with some value
VariableLengthData variableLengthData1(dataPointer1, data1Size);
void * dataPointer2; // fill data with some value
VariableLengthData variableLengthData2(dataPointer2, data2Size);
paramValues[parameter1Name] = variableLengthData1;
paramValues[parameter2Name] = variableLengthData2;
std::string tag("Any String");
rtiAmbassador->sendInteraction(
    theInterClassHandle,
    paramValues,
    VariableLengthData(tag.c_str(), tag.size()+1));
```

Listing 8 – Code snapshot for sending an interaction over HLA federation

Time Advance Request

```
LogicalTimeInterval* lookAhead = new LogicalTimeIntervalImpl(1.0f);
LogicalTime* currentRTITime = new LogicalTimeImpl(0.0);
LogicalTimeInterval* oneTimeStep = new LogicalTimeIntervalImpl(1.0f);
// first tell rti that the federate will be a time regulating and time constraint
rtiAmbassador->enableTimeRegulation((*lookAhead));
rtiAmbassador->enableTimeConstrained();
// first update currentRTITime time
(*currentRTITime) += (*oneTimeStep);
rtiAmbassador->timeAdvanceRequest(*currentRTITime);
```

Listing 9 – Code snapshot for enabling time regulating/constraint and requesting time advance over HLA federation

2.2.5 Terminating execution

Termination execution involves two steps, namely, resigning from a federation execution and destroying the federation execution if there are no other federates joined to the federation.

```
rtiAmbassador->resignFederationExecution(rti1516::DELETE_OBJECTS);

try
{
rtiAmbassador->destroyFederationExecution(federationName);
}
catch (FederatesCurrentlyJoined& ex)
{
    // Some federates is still joining the federation
    // ignore the destroy operation.
}
```

Listing 10 – Code snapshot for resigning from a federation and destroying a federation execution

2.3 Functional Mockup Interface (FMI)

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 [3] [4].

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 their solvers in them. As we can define HLA Federates as standalone simulation runnables, this effort utilizes FMI-Co-simulation interface as the basis for federate development.

As shown in Figure 1, FMU can be generated from different tools and for different systems/subsystem according to existent requirements.

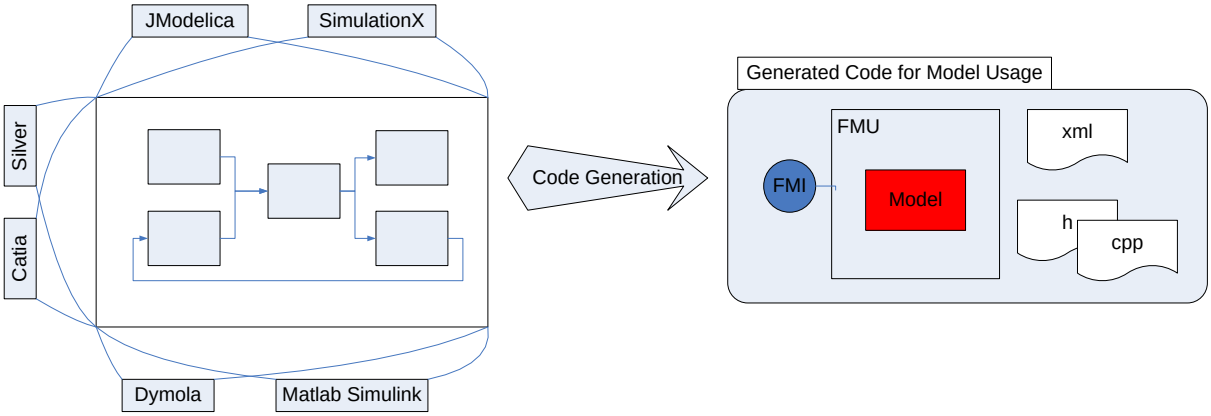

Figure 1 – Generating FMU from different tools

2.3.1 History

The FMI standardized interface was developed as one of the products of MODELISAR project. MODELISAR was an Information Technology for European Advancement (ITEA) project aiming to improve the design of systems and of embedded software in vehicles. The project was started in 2008 and finished in 2011. After it ended, FMI was managed and developed as a Modelica Association Project (MAP). On Jan. 2010, the version 1.0 of "FMI for Model Exchange", On Oct. 12, 2010, the version 1.0 of "FMI for Co-Simulation" and on Mar. 31, 2011, the version 1.0 of "FMI for PLM" were released [17].

In this thesis, the version 1.0 of "FMI for Co-simulation" is used as a reference FMI standard.

2.3.2 Definitions

Functional Mockup Interface

The FMI defines a standardized interface to be used in computer simulations for developing complex cyber-physical systems. FMI provides three aspects of creating models as listed below:

- FMI for model exchange,

- FMI for Co-simulation
- FMI for PLM

Functional Mockup Unit

A component implementing the FMI is called Functional Mock-up Unit (FMU). FMU is crated as a zip file. This file should contain a model description file. Besides, this zip file can also contain libraries, source codes, a model icon and documentation. The content of this zip file is shown at Figure 2.

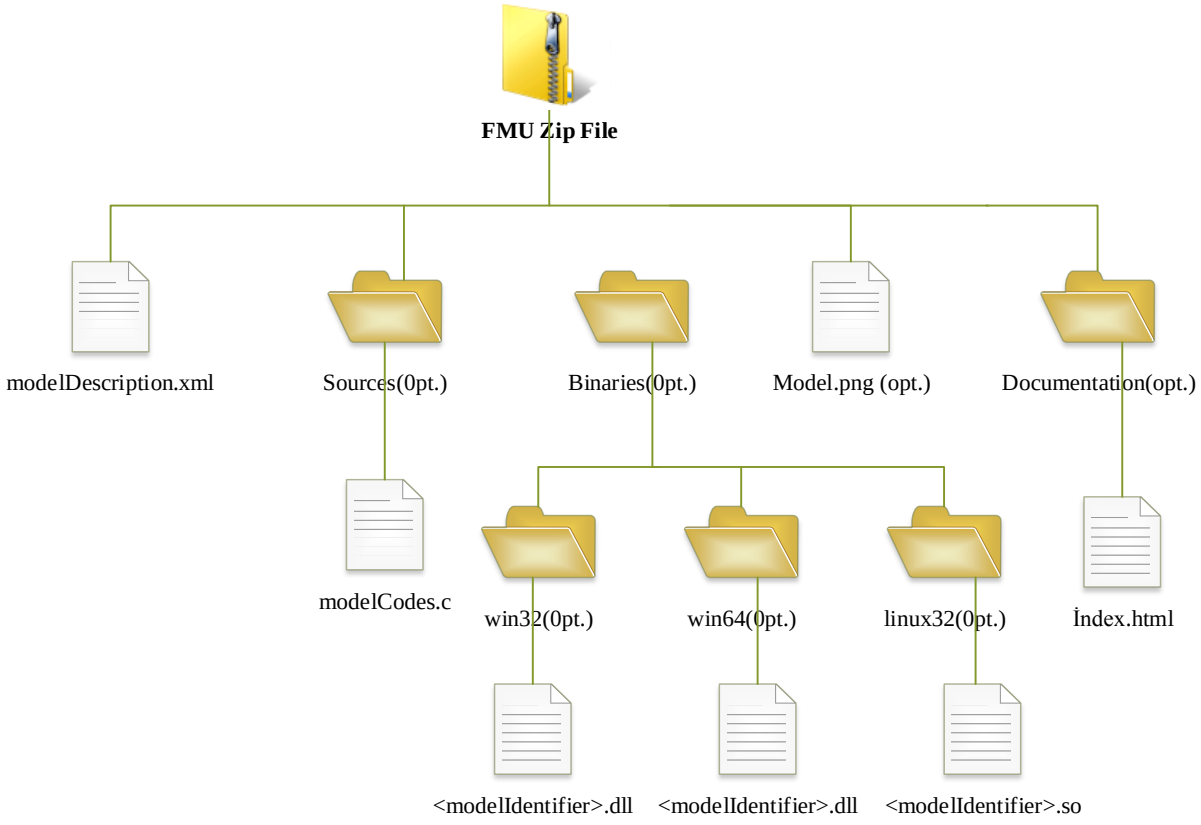

Figure 2 – The content of the FMU zip file

FMI for Model Exchange

FMI for model exchange is a standard interface that is a solution for the Ordinary Differential Equations of the models. The intention is to exchange of model among different modeling environments. A modeling environment can generate C-Code of a

dynamic system model that can be utilized by other modeling and simulation environments. As shown in Figure 3, the solver is not a part of the model, but it is a part of the environment that uses the FMU [3].

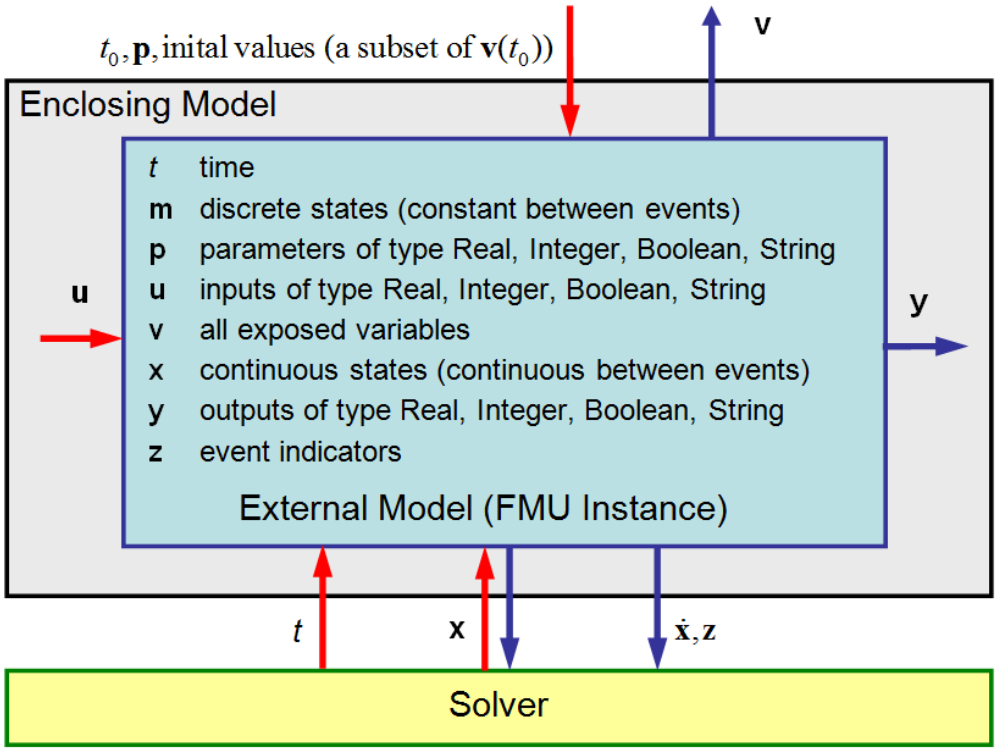

Figure 3 – Data flow between the environment and an FMU for Model Exchange [5]

FMI for Co-simulation

FMI for Co-simulation is a standard interface for the model output where the output contains its solver inside. Therefore, the FMU generated from FMI for co-simulation interface is acting as a black box for the user of the model. That is, the user does not need know which integration method is actually applied to solve the model. The user only feeds the model with inputs and gets the outputs from model in discrete time steps. Figure 4 shows the direction of the data in the FMI for Co-simulation standard [4].

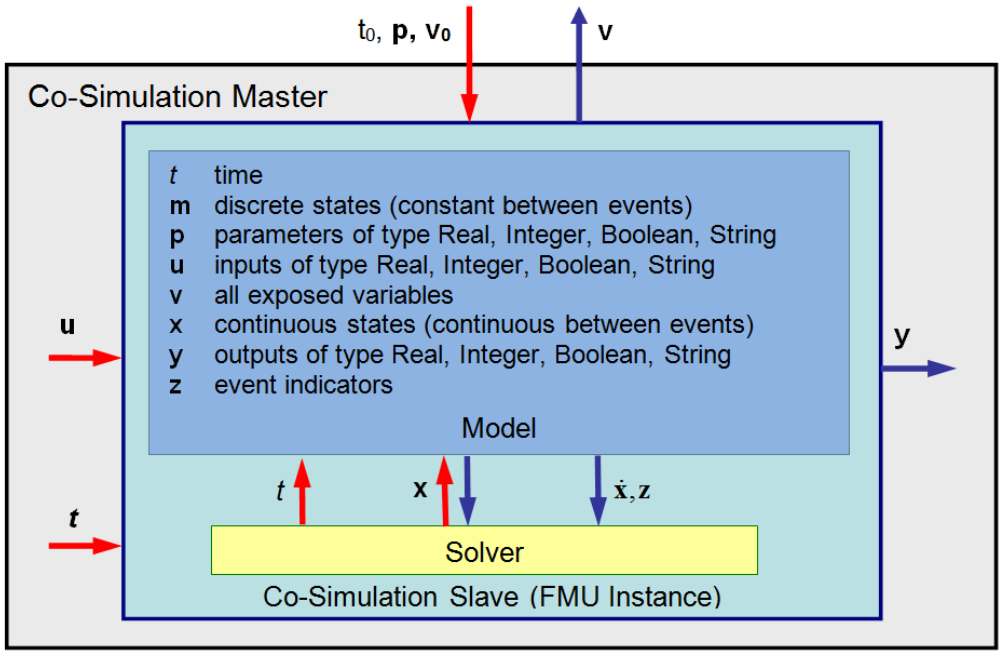

Figure 4 – Data flow between the environment and an FMU for Co-simulation at communication points [6]

FMI for PLM

FMI for Product Lifecycle Management (PLM) is an interface standard for supplying a generic way to manage the integration of FMI models and related data in product life-cycle management [17]. This management includes followings data:

- FMU data for documentation, simulation and validation
- Co-simulation data for documentation, simulation and result management
- Result validation data for post-processing, analysis, report FMU data

Model Description

Model description is an XML file which contains the static information of the FMU except the model equations. This file contains the model scalar variables and their attributes with such as name, unit, and default initial value etc. Model description XML structure is figured out in Figure 5 [3].

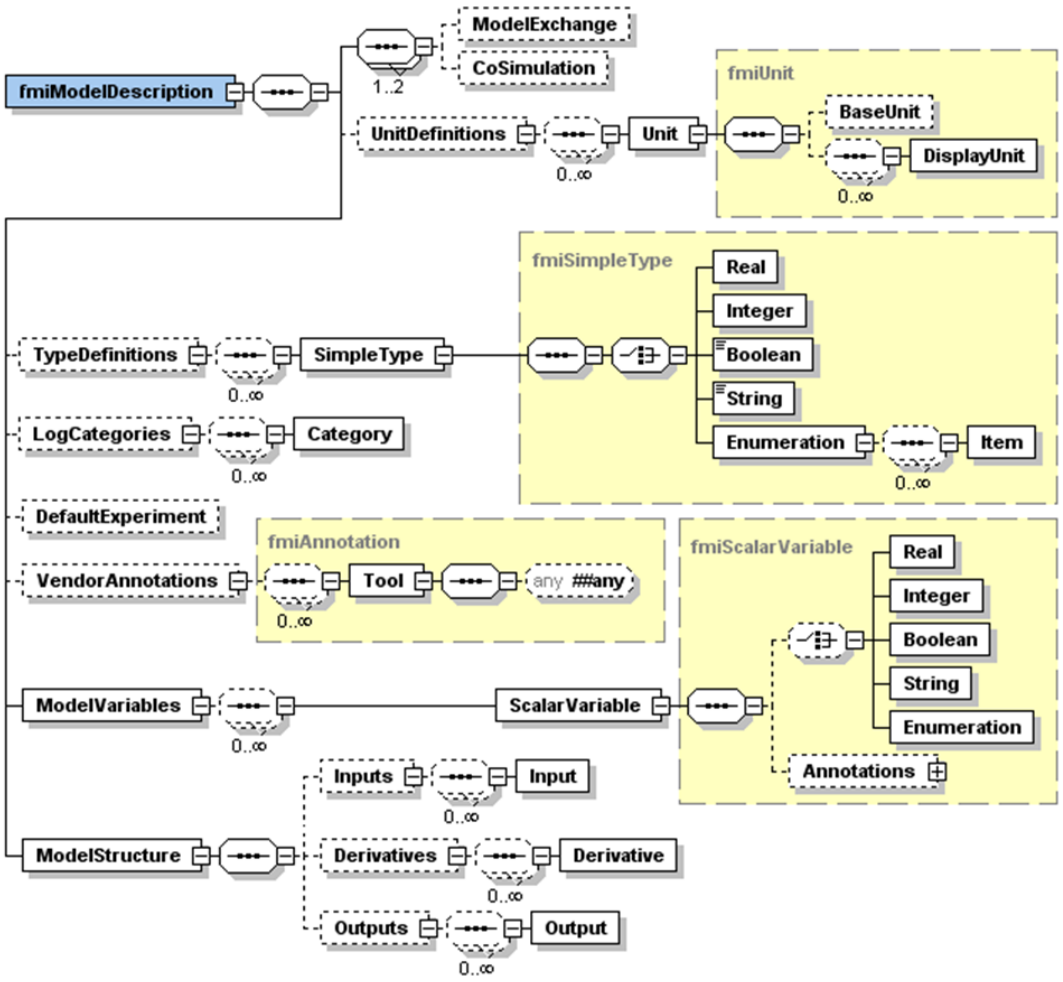

Figure 5 – Model description XML file structure [3]

2.3.3 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 applied 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 done in a discrete communication point, which is a sampling point or a synchronization point of the FMU [4].

2.3.4 Computational Flow

As show in Figure 6, FMU co-simulation computational flow has three main states, namely Instantiation and Initialization, Running and Termination.

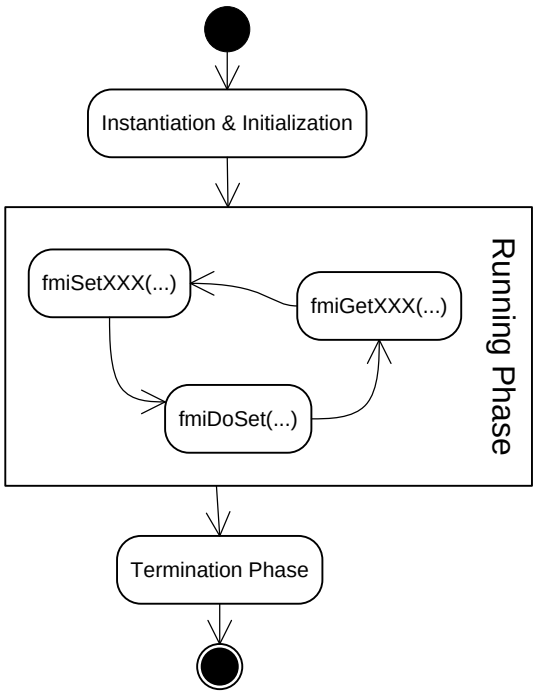

Figure 6 – The computational flow of the FMU for Co-simulation [5]

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 done within this phase.

Running

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

Termination

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

2.4 FMI for Co-simulation Programming in C

FMI for Co-simulation provides an interface for using FMU. This section briefly summarizes the calling of this interface.

Some parts of the information, which will be presented in this section, are extracted from FMI SDK [18].

Instantiate a Model Instance

FMU uses callback functions to utilize resources from the environment and to callback the function logs to the master application.

```
// the path for FMU file
 const char* fmuFileName = "FMU file path";
// the fmu to simulate
FMU fmu;
loadFMU(fmuFileName);
 // handle to the parsed XML file
ModelDescription* md = fmu.modelDescription;;
// instance of the fmu
fmiComponent c;
// the callback function
fmiCallbackFunctions callbacks;
// a unique identifier for a given FMI Component instance
const char* modelIdentifier = "instanceName";
// global unique id of the fmu
const char* guid = "{8c4e810f-3df3-4a00-8276-176fa3c9f003}";
// path to fmu
const char* fmuLocation;
// denotes tool in case of tool coupling
const char* mimeType = "application/x-fmu-sharedlibrary";
// wait period in milli seconds, 0 for unlimited wait period"
fmiReal timeout = 1000;
// no simulator user interface
fmiBoolean visible = fmiFalse;
 // simulation run without user interaction
fmiBoolean interactive = fmiFalse;
fmiBoolean loggingOn = fmiTrue;

void fmuLogger(fmiComponent c, fmiString instanceName, fmiStatus status,
    fmiString category, fmiString message, ...)
{
    // log the messages
}
callbacks.logger = fmuLogger;
// allocate memory method
callbacks.allocateMemory = calloc;
// free memory method
callbacks.freeMemory = free;
// fmiDoStep has to be carried out synchronously
callbacks.stepFinished = NULL;
c = fmu.instantiateSlave(getModelIdentifier(md), guid, fmuLocation, mimeType,
    timeout, visible, interactive, callbacks, loggingOn);
if (!c)
{
    // Error: could not instantiate the FMU
}
```

Listing 11 – Code snapshot for instantiating an FMU

Initialize a Model Instance

After instantiating the FMU, it needs to be initialized. In this section, the initial values of model are set inside the model.

```
// start time
double tStart = 0;
// end time
double endTime = 1000;
// StopTimeDefined=fmiFalse means: ignore value of tEnd
fmiBoolean StopTimeDefined = fmiTrue;
fmu.initializeSlave(c, tStart, StopTimeDefined, endTime);
```

Listing 12 – Code snapshot for initializing an FMU

Stepping

For every cycle, model can be stepped. By convention, before stepping the model, the input values of model is set and after step function is finished, the model output values are consumed by caller application.

```
// current simulation time
double currTime = startTime;
// the single step time
double stepTime;
// fill the array with scalar variable referances
fmiValueReference* valueRefs;
// fill the array with related scalar variables values
fmiReal* values;
while (currTime < endTime)
{
    // set values for scalar variables
    fmu.setReal(c, valueRefs,1, values);

    fmiStatus fmiFlag = fmu.doStep(c, currTime, stepTime, fmiTrue);
    if (fmiFlag != fmiOK)
    {
        //Error: could not complete simulation of the model
        // return error code
    }

    // set values from scalar variables
    fmu.getReal(c, valueRefs,1, values);

    currTime += stepTime;
}
```

Listing 13 – Code snapshot for setting scalar variables values, getting values of the scalar variables and stepping FMU over time

Ending Simulation

Whenever the simulation is completed, FMU is terminated and the allocated resources are freed into the memory.

```
fmiStatus fmiFlag = fmu.terminateSlave(c);
fmu.freeSlaveInstance(c);
```

Listing 14 – Code snapshot for terminating simulation and cleaning the used memory

CHAPTER 3

RELATED WORKS

As model based development of engineering systems are getting more and 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 four particular efforts providing a mechanism for connecting models to HLA environments.

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

In [10], the author introduces a concept to use HLA RTI as a FMI Co-simulation master. This concept defines how to use HLA RTI services as a master. These services are used as follows:

- Object Management: Only one HLA object class is enough to represent all FMUs. Each FMU (or hosting application) creates an instance of that object class and publishes related attributes. Likewise, it subscribes related attributes of the object class.

- Declaration Management: Hosting code can perform publishing and subscription generically, if some more directives are added into model-description file. These directives should guide the code about the place and role of each variable in the federation.

- Ownership Management: In normal cases, there is no need for ownership management; however, if more than one federate has the right to publish and

update a specific attribute, then there should be a separate module or FMU to overcome that scenario.

- Time management: RTI time management services can be used to synchronize the FMUs. The FMU can run a step between two time advances.

The effort does not provide a specific way to connect FMU to the HLA simulation environment. Our work, on the other hand, supplies a solution to connect FMU to the HLA environment with details. Moreover, the paper offers a special distributed simulation environment in which the FOM file is specialized for the FMUs. Our work; however, does not need a special simulation environment. Instead, it offers to connect any HLA distributed simulation environment with any FOM file.

In [12], the author states a simulation environment to develop a missile where a hardware-in-the-loop (HIL) simulation system based on high Level Architecture (HLA) and Modelica language are used. Although the concept is defined in paper, there is no detailed information about implementation of the simulation environment. Also, the work specifically focuses on developing a missile.

In [13], the author declares the SPRINT project which is a corporation of six European companies. The goal of the project is to simplify collaborative systems engineering across tools and platforms and validate the systems including physical devices based on distributed real time simulation using HLA and FMI. Although author states the goal of the project, there is no evidence of the status of the project.

CHAPTER 4

FUNCTIONAL MOCKUP UNIT FEDERATE DESIGN

FMI for Co-simulation standard does not provide a specification for connecting FMUs to an HLA federation. There is no convenient way to convert FMU scalar variables to HLA object class attributes, since 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. Moreover, FMI Co-simulation does not have suitable interface to consume HLA services as listed in section 2.1.4. FMU, instead, only supports some form of functions with primitive event.

Hence, there is a need for a wrapper connecting FMU that connects the 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 a FMU model.
- Providing the communication of distributed environment with services by using 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.

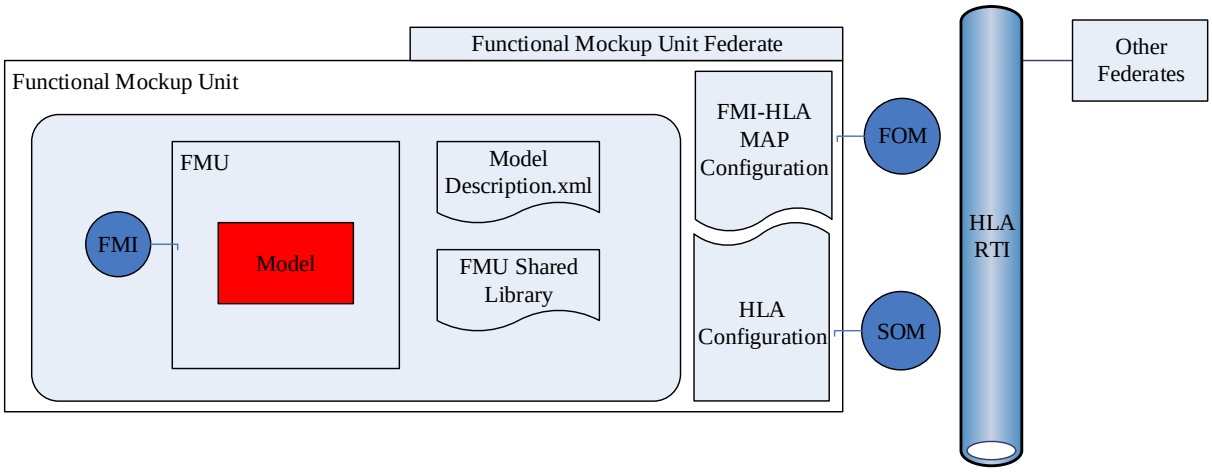

Figure 7 – The conventional design of Functional Mockup Unit Federate

The top level structure of FMUFd that satisfies these requirements is depicted in Figure 7. 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 8, these activities can be grouped into four main phases, namely, initialization, object discovery, stepping and termination.

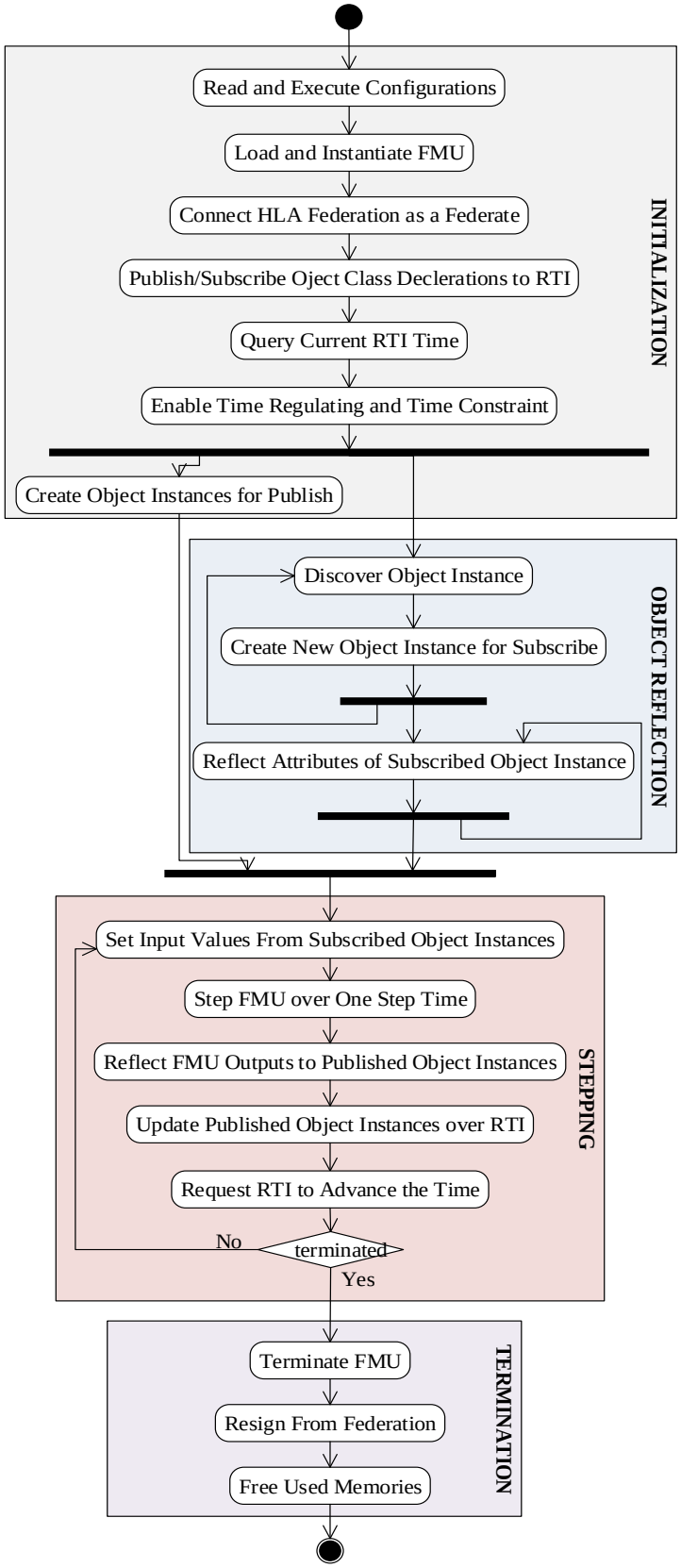

Figure 8 – FMUFd Activity Diagram

In **initialization** phase, FMUFd loads and initializes the FMU and then connects the HLA federation as a federate with related HLA services and declares 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 the 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 proceeding sections. Following 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 is designed to have the same name with model name and shared object FMI functions should also take the model name as a prefix to their functions [4].

The FMU related operations are developed based on the FMU SDK [18]. 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 Execution

HLA Federation is a named set of federate applications and a common Federation Object Model (FOM) that are used as a whole to achieve some specific objective. The Federation execution is, on the other hand, the actual operation, over time, of a set of joined federates that are interconnected by a RTI.

The FOM file contains all data exchange related information of HLA Federation, including object classes and object class attributes. By parsing 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 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 that 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

HLA Object instance is a unique instantiation of an object class that is independent of all other instances of that object class.

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 by 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 size. While calculating the size of a data type the padding rules are used as described in APPENDIX A - HLA DATA TYPE PADDING RULES. Although there are some existing 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 cause 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 a 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 Listing 15. 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(dataType->type == ObjectClass::Attribute::DataType::BasicData)
{
    int mod = dataType->size;
    int padding = (dataType->size - (currentOffset % mod)) % mod;
    dataType->offset = currentOffset + padding;
    currentOffset = newDataType->size + dataType->offset;
}
```

Listing 15 – The base condition code snapshot for calculating the padding bytes

4.2.3 Update/Reflect Object Class Attributes

Update/reflect object instance means invoking the Update/Reflect Attribute Values services of RTI for one or more instance attributes. By doing that, the attribute values can be updated from its owner federate and can be reflected by other federates.

The attribute contains values inside its data type. 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 prior to the call of update operation. 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 Listing 16. 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 == ObjectClass::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;
        }
    }
}
```

Listing 16 – The base condition code snapshot for encoding/decoding the attribute values.

4.3 Running the Federate

After introducing how HLA data is de-marshaled, 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 creation of an 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, 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 Implementation Details

The FMUFd is implemented as an application. This application can be run either as a terminal application or a desktop application.

The implementation details of the application are explained in the following sections.

4.4.1 The Layers of Application

Inspired from "Layered Simulation Architecture" paper [19], the application is constructed with three base layers as shown in Figure 9.

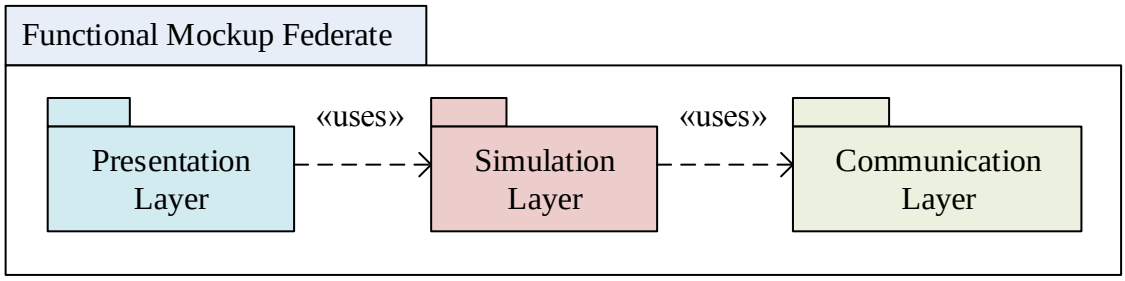

Figure 9 – The top level structure of FMUFd

4.4.1.1 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 Figure 10. 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.

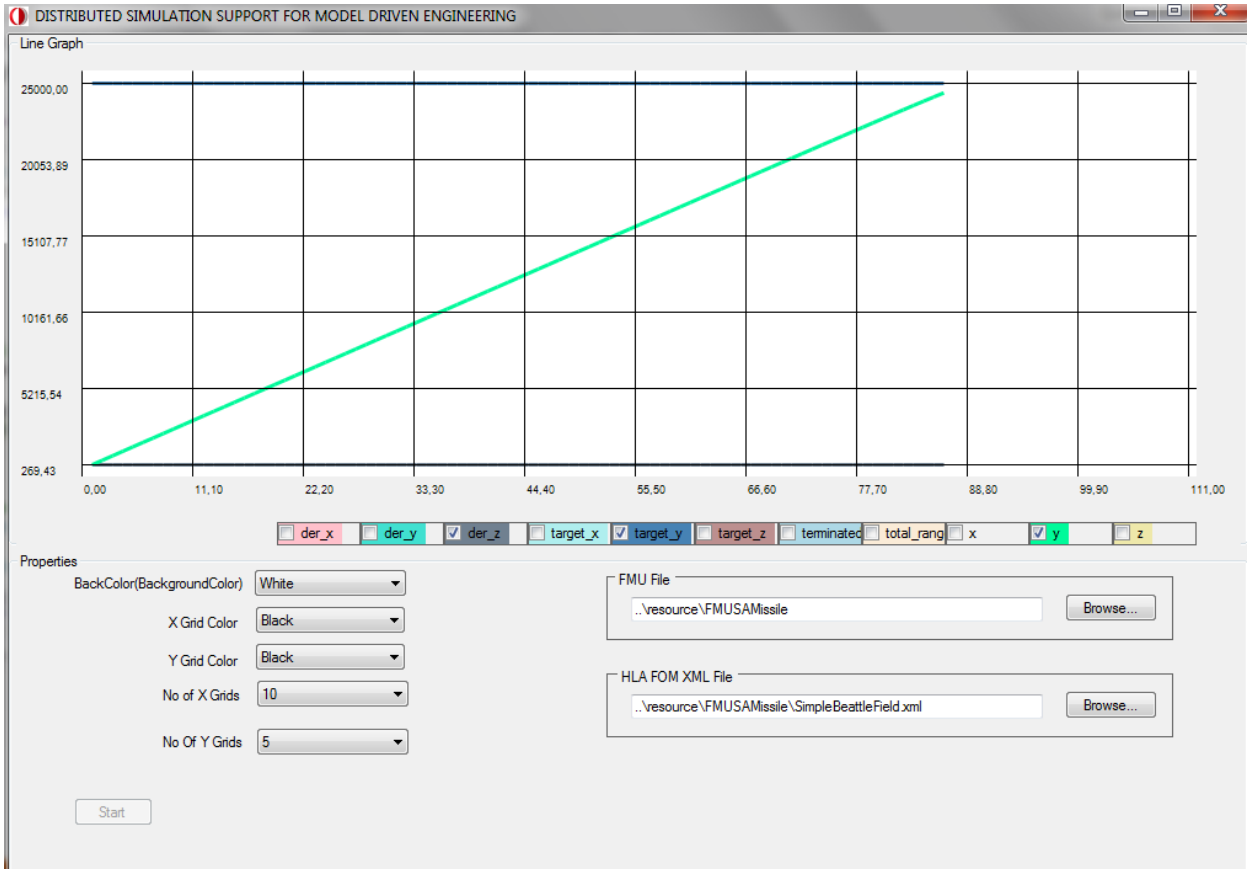

Figure 10 – The user interface of the FMUFd

4.4.1.2 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 model 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.

As shown in Figure 11, simulation layer is mainly composed of two packages: *Simulation Manager* and *FMIWrapper*.

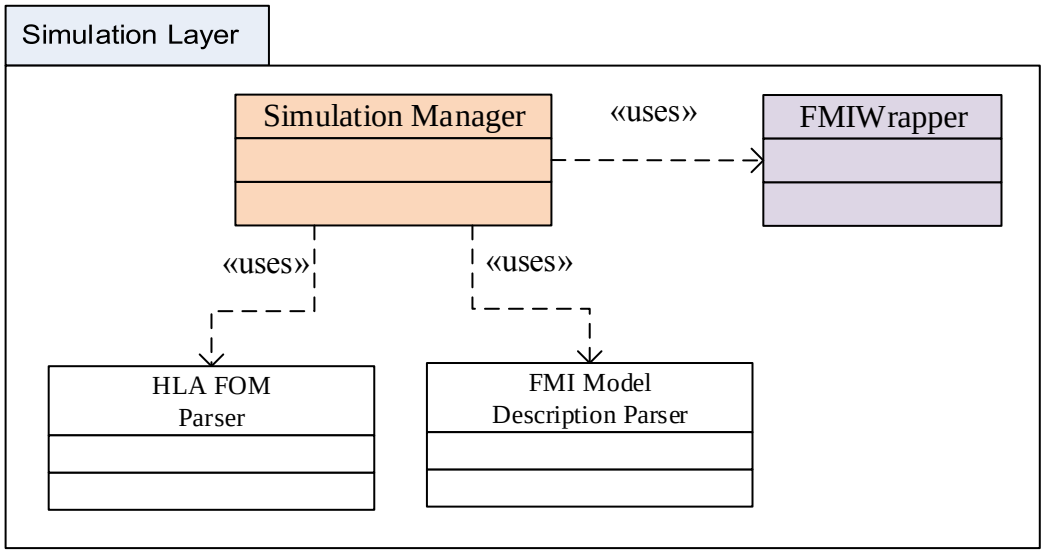

Figure 11 – Simulation layer class diagram

Simulation Manager Layer is the core simulation layer. This layer is responsible for controlling the simulation flow. This layer also loads the *FMIWrapper* layer and uses this layer to reach FMU. In addition, this layer is responsible for loading and using communication layer to communicate with HLA federation. Finally, this layer provides the feedback of the FMI scalar variable values over time to the presentation layer.

FMIWrapper layer is responsible for loading the FMU model instance and supplying the necessary interface functions to run FMU. *FMUWrapper* supplies necessary header files (namely, *fmifunctions.h* and *fmiPlatformTypes.h)* to provide the necessary platform information to the FMU. Also, *FMIWrapper* provides FMUs with the necessary memory allocation and de-allocation functions.

4.4.1.3 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.

The basic structure of communication layer is represented in Figure 12. The *MAKHLACommunicator* class is responsible for connecting the federation and publishing the HLA object classes. The *MAKHLAFederateAmbassador* class is responsible for discovering the class instances and receiving the updates. The *HLAClassContainer* is the container of object classes for both classes.

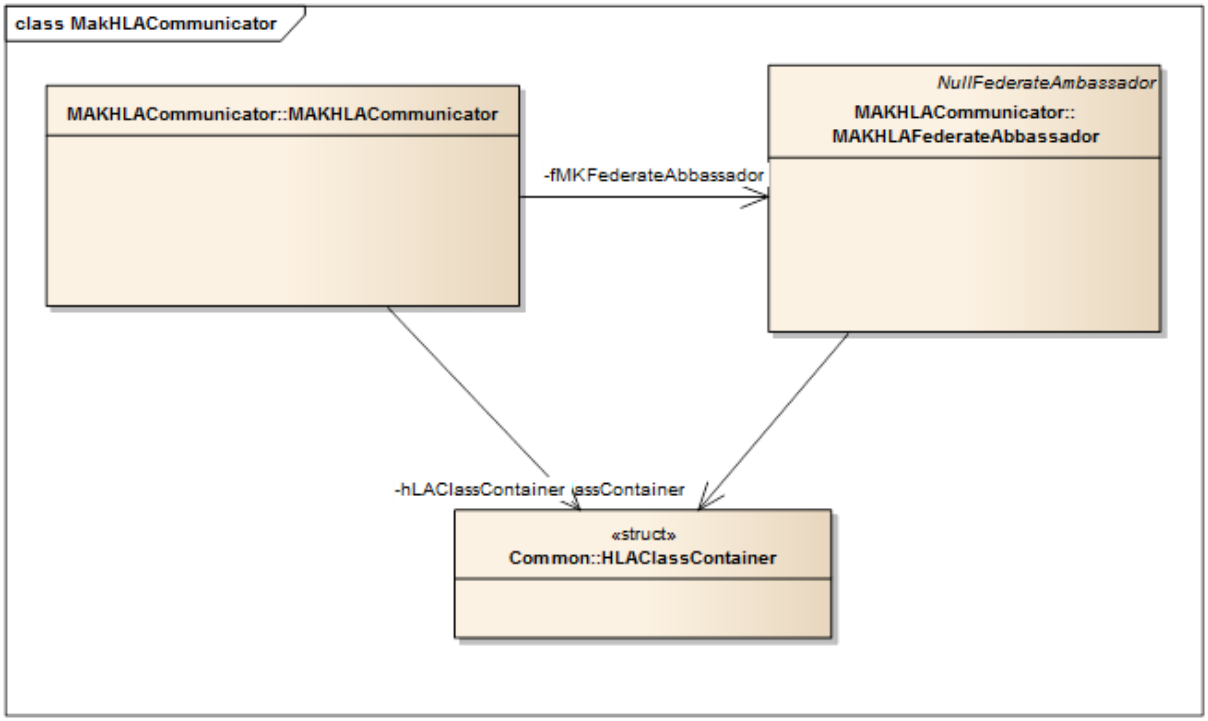

Figure 12 – Communication layer class diagram

4.4.2 Application Sequence Diagrams

Basically, there are two main phases of application: Initialization Phase and Simulation Phase. The initialization phase can be defined as a couple of operations run prior to the start of simulation. In the simulation phase, on the other hand, the application is run.

4.4.2.1 Initialization Sequence Diagram

As shown in Figure 13, simulation manager manages the application to the main simulation. First, simulation manager reads and interprets the configuration files. Second, helping with *FMIWrapper* package, it loads FMU and instantiates that model. Last, helping with *HLACommunicator* package, it connects a federation as a federate.

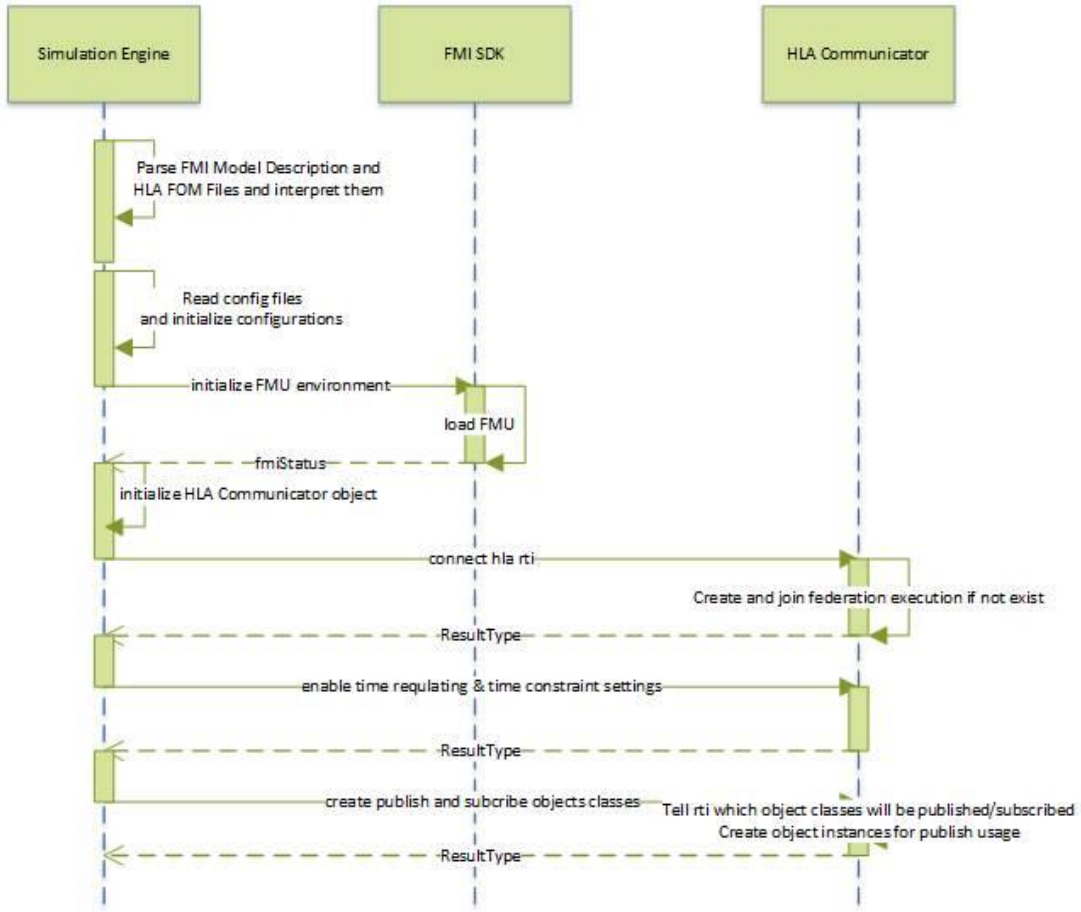

Figure 13 – Simulation initialization sequence diagram

4.4.2.2 Simulation Running Sequence Diagram

As represented in Figure 14, there is a loop for each step of simulation. In this loop, first, simulation manager feeds the FMU model inputs from related HLA object class attribute fields. Then, FMU runs for one step. After then, simulation manager, feeds related HLA object class attribute fields from FMU outputs and publishes those

changes over federation. Lastly, application requests the RTI to update simulation time and waits until request is fulfilled.

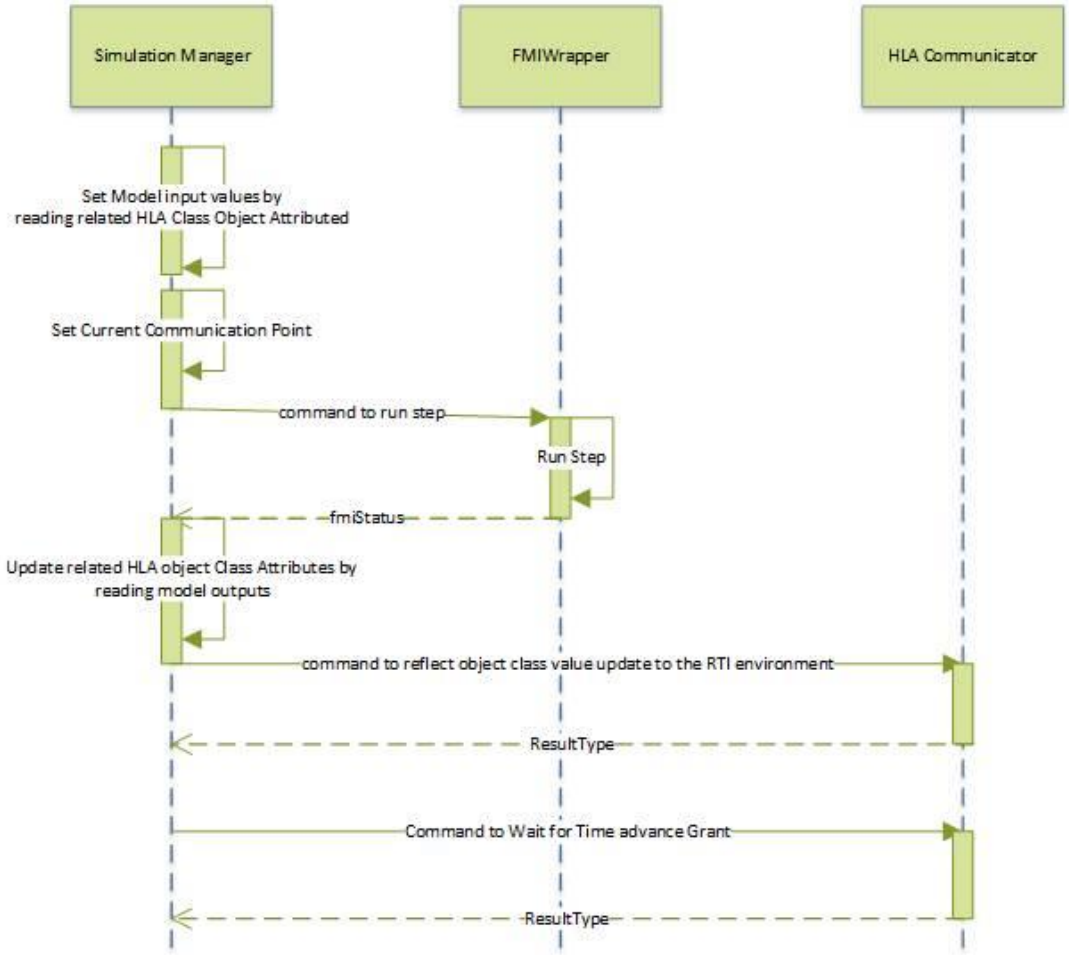

Figure 14 – Simulation running sequence diagram

4.5 FMUFd Capabilities

Here in this section, FMUFd capabilities are expressed 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 checks whether the object instance is discovered before. If it is discovered, FMUFd reflects the attribute values to the allocated memory of the object instance, and ignores otherwise. Whenever a *removeObjectInstance* event is raised by RTI, the FMUFd checks whether the object instance is discovered before. If it is discovered, FMUFd deletes the handle of instance and frees the related allocated memory.

FMUFd reflects the attribute values to the allocated memory of the object instance, and 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

There are some limitations about FMUFd as listed below:

- FMUFd does not support array data type with dynamic cardinality. If FMUFd would support this feature, it should have checked the size of attribute dynamically on every update. This would create some performance issues. Instead, FMUFd statically specifies the size of the attribute at the beginning of the simulation
- Interactions are not supported by FMUFd.
- As FMI standard does not support the float type, if there is a mapping between FMU double and HLA float data type, and then there could be loss of precision.
- As FMI standard does not support the short type, if there is a mapping between FMU integer and HLA short data type, then there could be loss of precision.

CHAPTER 5

DEMONSTRATION WITH AN EXAMPLE FEDERATION

To demonstrate the FMUFd usage, a simple distributed simulation environment is developed with MAK HLA RTI [15] implementation. For this application, the RPR2-D17 FOM file developed by SISO [20] is used as a FOM file.

5.1 Simulation Setup

There are three nodes connected over an Ethernet network in this distributed simulation environment as shown in Figure 15. In the missile node, the missile co-simulation FMU (called *MissileFMUFd*) is connected to the 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 [21].

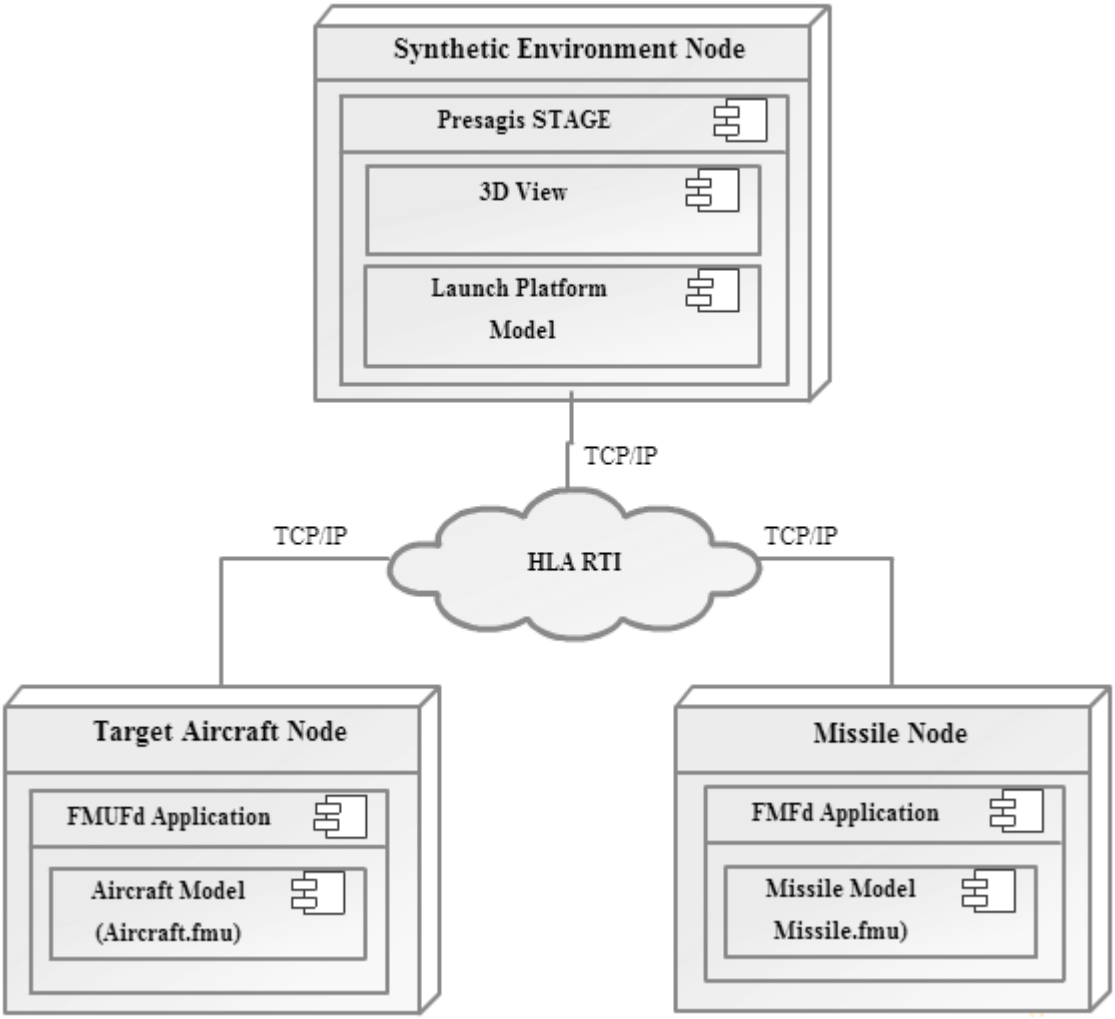

Figure 15 – The Deployment View Diagram of Simulation Environment

The MissileFMUFd application which will represent the missile FMU in the simulation will be run in Missile Node. To do that it needs a configuration directory as mentioned in APPENDIX B. The content of this directory is given in APPENDIX C.

The AircraftFMUFd application which will represent the target FMU model in the simulation will be run in Target Aircraft Node. To do that FMUFd needs a configuration file as mentioned in APPENDIX B. The content of this directory is given in APPENDIX D.

.

In synthetic environment node, federate is supplied by a third party tool named as STAGE [21]. This tool has a property of connecting HLA environment and showing the object models in 2D and 3D perspectives. We will use this tool to visualize the simulation environment. Also, this tool will publish a cultural feature object class to represent a launch platform model. Launcher itself does not affect the simulation; it will be only used for supplying the some of the input requirements of missile model.

5.2 Simulation Run & Result

The FMUFd federates running activities are exactly the same as shown in Figure 8. Because of the nature of the distributed simulation, each federate can start working in different times. After all the federates are crated, the missile runs towards the aircraft and finally it shuts the aircraft.

1500 simulation steps have been executed on this demonstration and there are two measurements calculated from this demonstration. 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. Measurements were taken on a computer with Intel Xeon 2.66GHz processor, 4GB DDR3 RAM and Windows 7 Pro 64 bit operating system.

The median time for updating FMU parameters from HLA objects is the consumed time in retrieving data from HLA object instances and passing these data to FMU input parameters. The median time for updating HLA attributes from FMU parameters, on the other hand, is the consumed time in retrieving output parameters data from FMU and passing these data to related HLA object instances.

CHAPTER 6

CONCLUSION

This thesis work represents a study to develop an environment for adapting FMUs which implement FMI Co-simulation interface to distributed simulation environment, specifically HLA environment. In order to achieve the goal, FMUs are wrapped as HLA federate with a particular mapping approach. The mapped FMUs are called as Functional Mockup Unit Federate (FMUFd). The wrapping approach to generate these FMUFds can be used for FMUs which implement FMI for Co-simulation version 1. Similarly, FMUFds can join to any HLA federation which supports IEEE Std 1516 standard.

In order to see the FMU wrapping approach in action, a simple distributed simulation environment is developed. In demonstration, two FMUFds are generated using the wrapping approach, namely *MissileFMUFd* and *AircraftFMUFd*, from two FMUs which implement FMI for Co-Simulation version 1. These generated FMUFds are connected to the distributed simulation environment as the HLA, specifically MAK HLA RTI [15].

With the FMUFd, a system model can be simulated as a part of an aggregate simulation of its operational setting. Therefore, the model can be tested, analyzed and evaluated from the beginning of development phases to final product. Moreover, this promotes a high level of reusability of system models supporting FMI. For example, missile model software can be run on the missile and, at the same time, this software can be used in a training simulator without any modification needed.

6.1 Prospects for Future Research

Although Functional Mockup Interface is a new standard in modeling technology, it appears to gather an increasing popularity and usage in many different fields all around the world. Distributed simulation environments are one of these areas of

usage. At this point, joining FMUs to HLA, which is a distributed simulation environment, is required. The work mentioned in this thesis shows that FMUs which implements FMI for Co-simulation can be manipulated to be able to join HLA environment. The developed approach supports MAK RTI vendor HLAs. Considering the expanding generality of the environments, wrapping approaches, which provides ability for FMUs to join other RTI vendors, should be developed.

On the other hand, considering the problem of joining FMUs to distributed simulation environments on its own, mapping FMUs to join other different distributed simulation environments such as Distributed Interactive Simulation (DIS) and Data Distribution Service (DDS) can be an attractive field of study.

In the work conducted, FMUFd appears as a layer between HLA environment and FMU. As an alternative to this approach, instead of layering the wrapping FMUs to join distributed simulation environment process, this layer can also be compacted in FMU itself. This approach comes up with generating FMUs with a specified system. At this point, prior to joining FMUs to HLA environment, a special framework used for FMU generation can be developed in order to ease and customize the process of FMU generation.

REFERENCES

[1] IEEE Standards Association. (2000). 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules.

[2] IEEE Standards Association. (2000). 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification.

[3] IEEE Standards Association. (2000). 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification.

[4] MODELISAR Consortium. (2010, January). Functional Mock-up Interface for Model Exchange Version 1.0. Retrieved from www.fmi-standard.org.

[5] MODELISAR Consortium. (2010, October). Functional Mock-up Interface for Co-Simulation Version 1.0. Retrieved from www.fmi-standard.org.

[6] Stenzel, C. (2008). Distributed Simulation in Technical Applications. Paper presented at X International PhD Workshop, OWD 2008, Conference Archives PETiS, Vol. Gliwice, Poland, 513-518.

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

[8] Stenzel, C., Pawletta, S. (2008). CERTI - Bindings to Matlab and Fortran. Retrieved from http://www.mb.hs-wismar.de/~stenzel/software/MatlabHLA.html

[9] HLA Toolbox™. The MATLAB® Interface to HLA. Retrieved from http://www.forwardsim.com

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

[11] Hollenbach, J. W. (2009). Inconsistency, Neglect, and Confusion; A Historical Review of DoD Distributed Simulation Architecture Policies. Florida, USA

[12] Bao, J., Feng, H., Wu, D., Lin, Y. (2011). Research on Hardware-In-the-Loop Missile Weapon System Simulation Based on HLA and Modelica. CHINA.

[13] Aronsson, P., Hedberg, D., Tronarp, O. (2012, October). A Collaborative Platform for Systems Engineering tools over the Internet. Retrieved from www.wolframmathcore.com.

[14] Saunders, R. (2000 to present). 1516 WG - HLA Evolved Working Group. Retrieved from http://standards.ieee.org/develop/wg/1516_WG.html

[15] MAK Technologies (2013). MAK RTI User's Guide

[16] MODELISAR Consortium (2012). Functional Mock-up Interface for Model Exchange and Co-simulation Version 2.0 Beta 4. Retrieved from www.fmi-standard.org

[17] MODELISAR Consortium (2011). FMI PLM Interface Version 1.0. Retrieved from www.fmi-standard.org

[18] QTronic Company Developer Team (2010). FMU SDK version 1.0.2. Retrieved from http://www.qtronic.de/en/fmusdk.html

[19] Topçu, O., Oğuztüzün, H. (2012) Layered Simulation Architecture: A practical approach. Ankara, TURKEY

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

[21] Stage Sales Team. (2013). Stage, A Complete Simulation Development Environment. Retrieved from http://www.presagis.com

APPENDIX A

HLA DATA TYPE PADDING RULES

HLA requires that certain types of data start at a particular 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 [2]:

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 structured 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:

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

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 and 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:

$$(Size + P) \bmod V = 0$$

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.

APPENDIX B

FMUFD CONFIGURATION DIRECTORY STRUCTURE

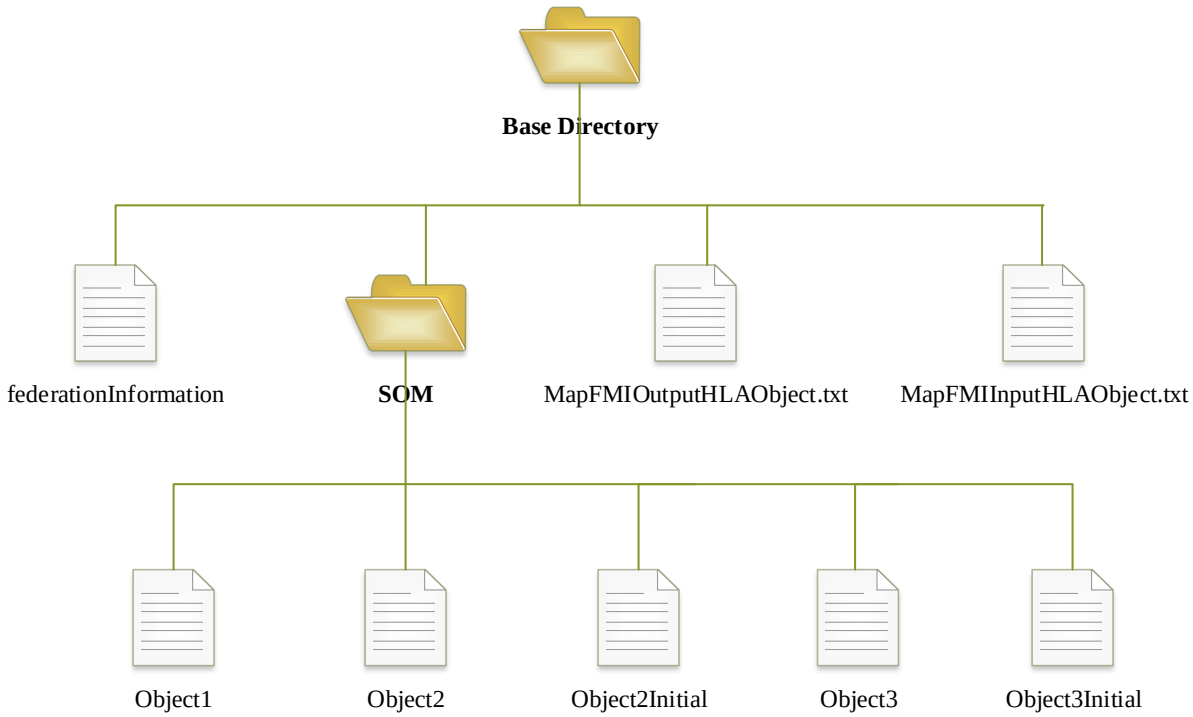

Figure 16 – The FMUFd Configuration Directory Structure

The FMUFd uses a configuration directory structure to map between FMU and Federation. The directory structure is given at Figure 16. The detail of the configuration structure is explained in following:

Base Directory

Base directory is the root of the configuration structure and its name can be anything. It should contain at least 3 files and one folder named as SOM.

FederationInformation

This file is used for informing FMUFd about federation information in which FMUFd will be connected to. The file should contain following information:

federationName = The Name of the fedation
federationFile = The path for the FOM XML file
federateName = The name of the FMUFd in federation

MapFMIInputHLAObject

This file is used for informing FMUFd to map between FMU input variables and HLA Object class's attribute's data type's sub data types' until it reaches the basic data type. The file structure should be given below:

ScalarVariable1 = ObjectClass|Attribute|DataType:DataType
:VariantRecordDataType[discriminant Name]:DataType…
ScalarVariable2 = ObjectClass|Attribute|DataType:DataType:DataType…

The left side of the equation represents the FMU object class name and right side of the equation represents the HLA object class's attribute's data type. The object class, attribute and data type are spitted with "|" character. A scalar variable can be mapped with only basic data type. If the datatype is complex then by using ":" character, its sub data types should be written until reaching the basic data type. If any of the data type represents the variant record data type, then the name of discriminant should be written between the square brackets.

Example usage of this file is given below:

Target_Ecef_X =
Aircraft|Spatial|SpatialStruct:DeadReckoningAlgorithm-A-
Alternatives:SpatialStructDeadReckoningAlgorithm[DRM_FPW]:SpatialFPS
truct
:WorldLocation:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways
:HLAfloat64BE

MapFMIOutputHLAObject

This file is used to inform FMUFd to map between FMU output variables and HLA Object class attribute's data type's sub data types until it reaches the basic data type. The file structure is given below:

ScalarVariable1 = ObjectClass|Attribute|DataType:DataType
:VariantRecordDataType[discriminant Name]:DataType…
ScalarVariable2 = ObjectClass|Attribute|DataType:DataType:DataType…

The left side of the equation represents the FMU object class name and right side of the equation represents the HLA object class' attribute's data type. The object class, attribute and data type are split with "|" character. A scalar variable can be mapped with only basic data type. If the datatype is a complex data type then by using ":" character, its sub data types should be written until reaching the basic data type. If any of the data type represents the variant record data type, then the name of discriminant should be written between the square brackets.

Example usage of this file is given below:

x = Aircraft|Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives
:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]
:SpatialFPStruct:WorldLocation
:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways:HLAfloat64BE

SOM

The directory named as SOM contains all the object class information as different files that will be used by FMUFd. By using these files, FMUFd learns which object class and its which attributes will be subscribed or published.

The SOM directory contains 2 types of files.

One type of the files represents object class. For every object class there should be a file with the same name with object class. This file's first line should contain the full path of the object class as object class can have same base classes. After that line, the below lines of @Publish or @Subscribe has tags will represent the published or subscribed attributes of that object class. Those attributes can be written in that class or its base classes. The file structure is given below:

The full path of the object class

@Publish

Attribute1

Attribute2

Attribute4

…

@Subscribe

Attribute2

Attribute3

Attribute5

…

For example, let's think about the Aircraft object class on RPR2-D17 FOM file developed by SISO [17]. The file name should be Aircraft and file content can be as follows:

HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.Aircraft

@Publish

EntityType

EntityIdentifier

Spatial

DamageState

FirePowerDisabled

FlamesPresent

ForceIdentifier

Marking

SmokePlumePresent

@Subscribe

EntityType

EntityIdentifier

Spatial

DamageState

FirePowerDisabled

FlamesPresent

ForceIdentifier

Marking

SmokePlumePresent

The SOM has another type of file which is named as object class name with “Initial” postfix. These files are optional and used for initializing the object class instance values. The structure of file is as follows:

ObjectClass|Attribute|DataType:DataType:VariantRecordDataType[discriminant Name]:DataType… = value1, value2, value3…

The datatype path doesn’t need to go until base datatype. Instead, values can be given with order of data type fields. For example, Aircraft object class (represented RPR2-D17 FOM file [17]) initial values file should be named as AircraftInitial and its contant can be as follows:

EntityType|EntityTypeStruct:EntityKind = 1

EntityType|EntityTypeStruct:Domain = 2

EntityType|EntityTypeStruct:CountryCode = 225

EntityType|EntityTypeStruct:Category = 1

EntityType|EntityTypeStruct:Subcategory = 3

EntityType|EntityTypeStruct:Specific = 3

EntityType|EntityTypeStruct:Extra = 0

EntityIdentifier|EntityIdentifierStruct:FederateIdentifier:FederateIdentifierStruct

:SiteID = 55

EntityIdentifier|EntityIdentifierStruct:FederateIdentifier:FederateIdentifierStruct

:ApplicationID = 65

EntityIdentifier|EntityIdentifierStruct:EntityNumber = 1

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]

:SpatialFPStruct:WorldLocation:WorldLocationStruct = 6379160, -111423, 3205.25

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-

DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:IsFrozen = 0

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-

DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:Orientation

:OrientationStruct = -1.0198171139, 0.0000000347, 2.2236523628

DamageState|DamageStatusEnum32 = 1

FirePowerDisabled|OMT13boolean = 0

FlamesPresent|OMT13boolean = 0

ForceIdentifier|ForceIdentifierEnum8 = 1

Marking|MarkingStruct = 1,'F','-','1','6'

SmokePlumePresent|OMT13boolean = 0

APPENDIX C

MISSILEFMUFD CONFIGURATION DIRECTORY

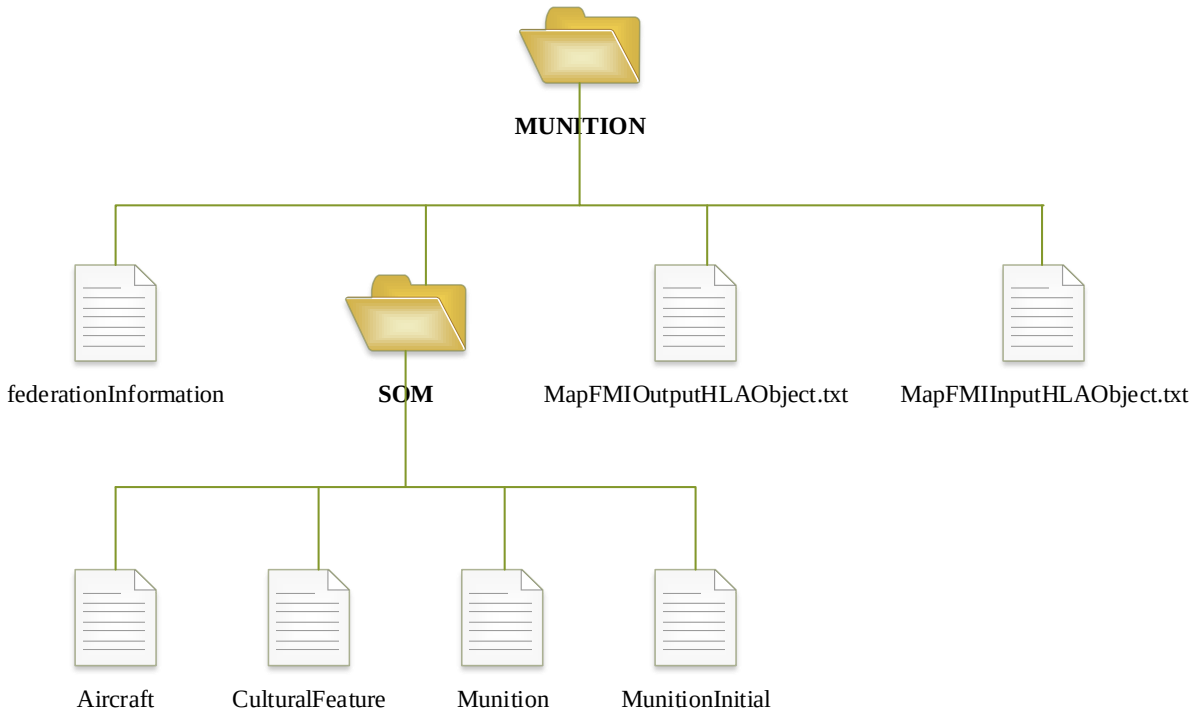

Figure 17 – *MissileFMUFd* Configuration Directory Structure

FMUFd which will use the directory structure shown in Figure 17 to represent the monition of FMU model in distributed simulation environment. The content of this configuration structure is given below:

FederationInformation

federationName = Battlefield

federationFile = ../resource/MissileProject/RPR2-D17.xml

federateName = MunitionFederate

MapFMIInputHLAObject

Target_Ecef_X = Aircraft|Spatial|SpatialStruct

:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways:HLAfloat64BE
Target_Ecef_Y= Aircraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:Y:HLAfloat64BEmetersperfectalways:HLAfloat64BE
Target_Ecef_Z= Aircraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:Z:HLAfloat64BEmetersperfectalways:HLAfloat64BE
Launcher_Ecef_X = CulturalFeature|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways:HLAfloat64BE
Launcher_Ecef_Y = CulturalFeature|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives
:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:Y:HLAfloat64BEmetersperfectalways:HLAfloat64BE
Launcher_Ecef_Z = CulturalFeature|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation
:WorldLocationStruct:Z:HLAfloat64BEmetersperfectalways:HLAfloat64BE

MapFMIOutputHLAObject

Missile_Ecef_X = Munition|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives
:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct
:WorldLocation:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways
:HLAfloat64BE
Missile_Ecef_Y = Munition|Spatial|SpatialStruct

:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct

:WorldLocation:WorldLocationStruct:Y:HLAfloat64BEmetersperfectalways

:HLAfloat64BE

Missile_Ecef_Z = Munition|Spatial|SpatialStruct

:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-

DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct

:WorldLocation:WorldLocationStruct:Z:HLAfloat64BEmetersperfectalways

:HLAfloat64BE

SOM

Aircraft:

HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.Aircraft

@Subscribe

EntityType

EntityIdentifier

Spatial

DamageState

FirePowerDisabled

FlamesPresent

ForceIdentifier

Marking

SmokePlumePresent

CulturalFeature:

HLAobjectRoot.BaseEntity.PhysicalEntity.CulturalFeature

@Subscribe

EntityType

EntityIdentifier

Spatial

Munition:

HLAobjectRoot.BaseEntity.PhysicalEntity.Munition

@Publish

EntityType

EntityIdentifier

Spatial

DamageState

FirePowerDisabled

FlamesPresent

ForceIdentifier

Immobilized

Marking

SmokePlumePresent

Munition:

EntityType|EntityTypeStruct = 2, 1, 225, 1, 1, 13, 186

EntityIdentifier|EntityIdentifierStruct = 55, 65, 2

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct

:WorldLocation:WorldLocationStruct = 6379160, -111423, 3205.25

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-

DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:IsFrozen = 0

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives

:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct

:Orientation:OrientationStruct = -1.0198171139, 0.0000000347,

2.2236523628

DamageState|DamageStatusEnum32 = 1

FirePowerDisabled|OMT13boolean = 0

FlamesPresent|OMT13boolean = 0

ForceIdentifier|ForceIdentifierEnum8 = 1

Immobilized|OMT13boolean = 0

Marking|MarkingStruct = 1,'f','u','z','e'

SmokePlumePresent|OMT13boolean = 0

APPENDIX D

TARGETFMUFD CONFIGURATION DIRECTORY

TargetFMUFd represents a target aircraft model which is simply flying with constant velocity and direction.

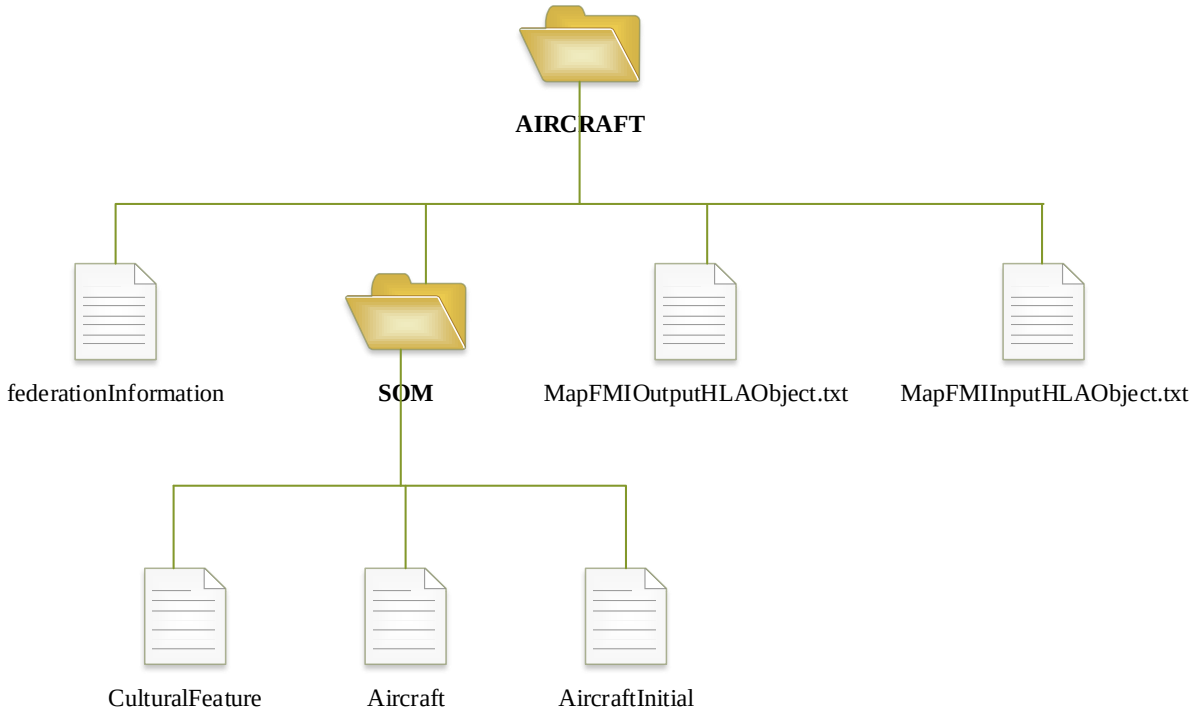

Figure 18 – *AircraftFMUFd* Configuration Directory Structure

AircraftFMUFd will use the directory structure shown in Figure 18 to represent aircraft FMU model in distributed simulation environment. The content of this configuration structure is given below:

FederationInformation

federationName = Battlefield

federationFile = ../resource/MissileProject/RPR2-D17.xml

federateName = TargetFederate

MapFMIInputHLAObject

Empty file

MapFMIOutputHLAObject

Ecef_X = Aircraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct
:WorldLocation:WorldLocationStruct:X:HLAfloat64BEmetersperfectalways
:HLAfloat64BE
Ecef_Y = Aircraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruc
t:WorldLocation:WorldLocationStruct:Y:HLAfloat64BEmetersperfectalways
:HLAfloat64BE
Ecef_Z = Aircraft|Spatial|SpatialStruct
:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-
DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct
:WorldLocation:WorldLocationStruct:Z:HLAfloat64BEmetersperfectalways
:HLAfloat64BE

SOM

Aircraft:

HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.Aircraft
@Publish
EntityType
EntityIdentifier
Spatial
DamageState
FirePowerDisabled
FlamesPresent
ForceIdentifier
Marking
SmokePlumePresent
@Subscribe

EntityType

EntityIdentifier

Spatial

DamageState

FirePowerDisabled

FlamesPresent

ForceIdentifier

Marking

SmokePlumePresent

AircraftInitial:

EntityType|EntityTypeStruct:EntityKind = 1

EntityType|EntityTypeStruct:Domain = 2

EntityType|EntityTypeStruct:CountryCode = 225

EntityType|EntityTypeStruct:Category = 1

EntityType|EntityTypeStruct:Subcategory = 3

EntityType|EntityTypeStruct:Specific = 3

EntityType|EntityTypeStruct:Extra = 0

EntityIdentifier|EntityIdentifierStruct:FederateIdentifier:FederateIdentifierStruct:SiteID = 55

EntityIdentifier|EntityIdentifierStruct:FederateIdentifier:FederateIdentifierStruct:ApplicationID = 65

EntityIdentifier|EntityIdentifierStruct:EntityNumber = 1

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:WorldLocation:WorldLocationStruct = 6379160, -111423, 3205.25

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:IsFrozen = 0

Spatial|SpatialStruct:DeadReckoningAlgorithm-A-Alternatives:SpatialStruct-DeadReckoningAlgorithm[DRM_FPW]:SpatialFPStruct:Orientation:OrientationStruct = -1.0198171139, 0.0000000347, 2.2236523628

DamageState|DamageStatusEnum32 = 1

FirePowerDisabled|OMT13boolean = 0

FlamesPresent|OMT13boolean = 0

ForceIdentifier|ForceIdentifierEnum8 = 1

Marking|MarkingStruct = 1,'t','a','r','g','e', 't'

SmokePlumePresent|OMT13boolean = 0