AN OBJECT-ORIENTED FRAMEWORK FOR FUNCTIONAL MOCK-UP INTERFACE CO-SIMULATION

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

BY

MEMDUHA ASLAN

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

FEBRUARY 2016

Approval of the thesis:

AN OBJECT-ORIENTED FRAMEWORK FOR FUNCTIONAL MOCK-UP INTERFACE CO-SIMULATION

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

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

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

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

Dr. Umut Durak
Co-supervisor**, Inst. of Flight Systems, DLR, Germany** _____

Examining Committee Members:

Prof. Dr. Ali H. Doğru
Computer Engineering Dept., METU _____

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

Assoc. Prof. Dr. Uluç Saranlı
Computer Engineering Dept., METU _____

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

Assist. Prof. Dr. Deniz Çetinkaya
Software Engineering Dept., Atılım Uni. _____

Date: 03.02.2016

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: Memduha Aslan

Signature :

ABSTRACT

AN OBJECT-ORIENTED FRAMEWORK FOR FUNCTIONAL MOCK-UP INTERFACE CO-SIMULATION

Aslan, Memduha

M.S., Department of Computer Engineering

Supervisor: Prof. Dr. Halit Oğuztüzün

Co-Supervisor: Dr. Umut Durak

February 2016, 89 Pages

Integration of subsystem or component models in a simulation environment is a crucial task in system development. Variation in model interfaces due to the use of a variety of modeling tools from different vendors complicates this task. Functional Mockup Interface (FMI) is a standard that lays out a tool-independent interface for dynamic system models. In practice, the developers of FMI-compliant models, known as Functional Mockup Units (FMUs), need guidance. In this thesis, an object-oriented framework for FMU development, integration and co-simulation, named MOKA, is introduced. MOKA Framework design maps the FMI 2.0 co-simulation specification faithfully to a set of class interfaces in C++. MOKA supports integration of FMUs developed using other tools as well. This thesis demonstrates the use of MOKA on a realistic case study.

Keywords: Functional Mockup Interface; Object Oriented Application Frameworks; Co-simulation

ÖZ

İŞLEVSEL MAKET ARAYÜZÜ İÇİN NESNE YÖNELİMLİ EŞ-BENZETİM ÇATISI

Aslan, Memduha

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Ortak Tez Yöneticisi: Dr. Umut Durak

Şubat 2016, 89 Sayfa

Alt sistemlerin veya sistem bileşenlerinin entegrasyonu, sistem geliştirme süreçlerinin ilk aşamalarında dahi oldukça önem taşır. Farklı üreticilerin sağladığı çeşitli modelleme araçlarının kullanımından kaynaklanan model arayüzlerindeki çeşitlilik, alt sistem modellerinin sıkı bağımlılığa sahip entegrasyonunu karmaşıklaştırır. İşlevsel Maket Arayüzü (İMA) dinamik sistem modelleri için geliştirme aracından bağımsız bir arayüz tanımlama amacıyla oluşturulmuş bir standarttır. Pratikte, geliştiriciler, İşlevsel Maket Birimi (İMB) adı verilen, İMA standardına uyan modellerin oluşturulması için bir kılavuza ihtiyaç duymaktadırlar. Bu tez, MOKA adı verilen ve İMB'lerin geliştirilmesi, entegrasyonu ve eş-benzetimi için ortam sağlayan nesne yönelimli bir yazılım çatısını sunmaktadır. MOKA yazılım çatısının tasarımı, İMA Eş-benzetim 2.0 versiyonunu standarda bağlı kalarak bir dizi C++ arayüzüne sahip sınıflar kümesine aktarır. MOKA aynı zamanda diğer modelleme araçları ile geliştirilmiş olan İMB'lerin de entegrasyonuna olanak sağlar. Bu tezde MOKA yazılım çatısının gerçekçi bir örnek senaryo üzerinde gösterimi de sunulmuştur.

Anahtar Kelimeler: İşlevsel Model Arayüzü; Nesne Yönelimli Yazılım Çatısı; Eş-Benzetim

To My Parents and Sisters

ACKNOWLEDGEMENTS

I would like to thank my supervisor 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 valuable comments and criticism for both thesis and demonstration work.

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

I would like to thank to gratefully acknowledge Faruk Yılmaz for their comments and help on case study development for this research.

I would like to thank to Ceren Aslan for all her invaluable support to complete this thesis work.

My special thanks are due to Uğur Taşdelen for his technical support, academic advice, understanding, endless patience and invaluable friendship.

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

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF LISTINGS

LISTINGS

LIST OF ABBREVIATIONS

API	Application Programming Interface
AVL	Anstalt für Verbrennungskraftmaschinen List
CENG	Computer Engineering
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DoD	Department of Defense
FMI	Functional Mockup Interface
FMIL	FMI Library
FMU	Functional Mockup Unit
GUID	Globally Unique Identifier
ITEA2	Information Technology for European Advancement 2
JFMI	Java FMI
MAP	Modelica Association Project
MBE	Model Based Engineering
METU	Middle East Technical University
MOKA	Model (Yeniden) Kullanım Altyapısı (Model (Re)use Infrastructure)
PLM	Product Lifecycle Management
SDK	Software Development Kit

CHAPTER 1

INTRODUCTION

The model based paradigm utilizes models as primary artifacts and employs them for the construction of engineering artifacts via (semi)automatic generation. Model-Based Engineering (MBE) proposes practicing model-driven paradigm pragmatically, not necessarily in an integrated fashion, in various steps of the engineering process [1].

Recent model-based approaches highlight the power of system simulation in different stages of the engineering processes. System simulation plays the role of real testing under certain conditions and supports understanding of system behavior particularly in early stages of development process. Therefore, system simulation is used in different phases of product development and gains importance throughout the product development cycle [2].

As the physical subsystems, models employed by different stages of system development processes are needed to be integrated for the interfacing with the operating environment in order to realize system simulation. Models for the various steps of the engineering processes are constructed by specialized development experts with the help of particular model development tools. Coordination of these tool sets and interfaces of models for seamless integration and simulation is always considered challenging.

Functional Mockup Interface (FMI) specification is a recent model interface standardization effort which is developed by a large industry consortium. The standard is aimed to be a tool-independent interface for integrating and simulating dynamic system models. A system model that complies with to this standard is called a Functional Mockup Unit (FMU). Depending on the nature of the system simulation environment, the interface supports two use cases: model exchange and co-simulation. In model exchange, FMUs that were exported from one simulation tool

set can be imported and simulated by the other one that supports FMI standard by using the solvers of the host tool set. This enables the models of one simulation environment to be available in the modeling and simulation environment of the other one. In the co-simulation, FMUs are exported in a form in which they can be simulated using their own embedded solvers. Hence, the simulation engineer only requires an FMU master to co-simulate multiple models from various sources. Providing such use cases for different purposes of simulation environments, FMI is well received by the industry and has already been implemented by various simulation tool sets [3][4].

FMI specifies a standard interface for dynamic system models but it does not guide the developer in terms of FMU construction and master development for coordination of the simulation. Hence, for the development of FMUs and co-simulation master model for co-simulation with standard modeling interface that is provided by FMI specification, developers need guidance and an infrastructure. Additionally, since FMI specification aims to support portability and utilization in a wide range of target platforms, widely-recognized software development approaches such as object-oriented development are left out from the specification. These issues are beyond the purview of FMI standardization. On the other hand, MBE can leverage object-oriented development approaches for modeling and simulation of the subsystems, in order to achieve more flexibility among components. Hence, the modeling and simulation community can benefit from the object-oriented approach to develop systems consisting of FMUs.

Existing well-established modeling and simulation tools already began to support the standard by providing interfaces to import FMUs to the simulation environment as well as to export developed models as FMUs. To illustrate, the FMI Toolbox for MATLAB/Simulink® from Modelon [5] that is a proprietary toolbox and enables FMU import/export for MATLAB/Simulink® for both model exchange and co-simulation. Such extensions enable the user to benefit from existing modeling and simulation environment with FMU models included. On the other hand, many

services for FMU development and integration such as manageable implementation of FMI interface for construction of FMUs and master FMU development are left out.

At this point, in addition to the applications added to existing development tools, there are some libraries which are adopted specifically to interact with FMUs beyond import/export. FMI Library (FMIL) by JModelica [6] and FMU Software Development Kit by QTronic [7] are the most popular libraries for developing and simulation of FMUs. In addition to such applications that provide a starting point on interacting with FMUs; there are also efforts that aim to extend FMI specification for object-oriented languages. The FMI++ Library [8] and Java FMI (JFMI) [18] are examples of such efforts to meet FMI specification with object-oriented languages.

Such existing approaches to interact with FMUs are needed to be enhanced in terms of different aspects. First of all, the FMI specification should not be touched so that the benefits of standardization can be secured. Secondly, FMU developers are needed to be guided on the development of FMUs and co-simulation master. Finally, FMU developers should be supported on the implementation of the FMI specification.

In this work, we address the need for an infrastructure to develop, integrate and co-simulate models that comply with the FMI standard. We find it efficacious to develop, integrate and co-simulate FMUs with framework completion. For this purpose, we introduce a modeling and simulation framework that extends the ideas introduced by such libraries like QTronic FMU SDK [7], FMI++ Library [8] , Java FMI (JFMI) [18] and aims to combine FMU development, master FMU construction and co-simulation capabilities. This framework, MOKA, is designed to serve as an overall simulation framework that provides a modular structure for FMUs and extendable co-simulation capabilities for integrating and executing multiple FMUs to perform a scenario. This approach has a potential to make the process of developing and co-simulating FMUs easier with a view towards maintenance and extendibility. To bridge the gap between the FMI specification and object-oriented development approaches, a modular approach for FMUs is provided in C++ language. While

achieving this, MOKA framework keeps the standard interface of the FMI specification to interact with FMUs.

MOKA Framework demonstrated and evaluated in terms of several aspects. In order to test the functionality and usage of the framework, a co-simulation environment which presents an engagement scenario that consists of a flying target and a guided missile with a seeker is developed and the results are evaluated. The case study co-simulation scenario with the MOKA Framework shows the promise to support the requirement for an infrastructure to develop, integrate and co-simulation of FMUs. The framework is further analyzed in terms of FMU development process and framework overhead. For the evaluation of FMU development process, the proposed development process is compared to the manual construction process for an FMU. An examination of this comparison suggests that MOKA Framework has the potential to promote code reuse and hide the details of FMU development process while guiding the developer. Finally, the execution time overhead caused by the framework is assessed in order to show how to calculate the execution time overhead of using the framework in a particular execution environment.

Preliminary results of the thesis work were presented in a conference paper entitled "MOKA: An Object-Oriented Framework for FMI Co-Simulation" in the Summer Simulation Multi Conference 2015 [24].

This thesis is organized as follows:

- In Chapter 2, background information for related topic of the study is given. Since the study introduces solution as framework completion, general structure of object-oriented frameworks is discussed. Co-simulation topic is also covered since the scope of the framework is FMI Co-simulation usage. Also, the FMI specification is discussed briefly and programming highlights for FMI Co-simulation is given.

- In Chapter 3, related works are discussed and similarities/differences from the MOKA Framework are pointed out.

- In Chapter 4, details of the MOKA Framework are presented. Firstly, this chapter introduces the overall structure of the framework. Then, implementation details are given and the mapping of FMI interface to a C++ API is elaborated as well as the object-oriented structure of the framework. The design rationale for the MOKA Framework is also given in this chapter.

- In Chapter 5, the usage of the framework is discussed in three sections: constructing an FMU, constructing an FMUMaster and co-simulation of various FMUs.

- In Chapter 6, an example simulation environment is constructed with the MOKA Framework. Firstly, the demonstration introduces the dynamic simulation models. After that, it describes the process of importing packaged standard FMU models that come from another development tool. Then, constructing remaining models via the MOKA Framework is shown. After that, integration of these models is described. Finally, co-simulation results are discussed. FMU Development process comparison and framework overhead assessment are also covered in this chapter.

- In Chapter 7, concluding remarks are provided. Future research directions to extend the presented work are pointed out.

CHAPTER 2

BACKGROUND

This chapter presents background for the research presented in this thesis. First of all, literature review for the general characteristics of object-oriented frameworks is presented. Secondly, co-simulation is reviewed, and regular co-simulation flow is presented. After that, FMI specification is presented in terms of history of the standardization, FMU structure, co-simulation in FMI and computational flow of FMU in a co-simulation scenario. Finally, the main programming steps for the co-simulation of an FMU are reviewed.

2.1 Object Oriented Frameworks

As the recent cyber-physical systems become more complex in terms of software components, the issue of developing software that is correct, efficient and portable needs more effort. Hence, main focus has become to promote developing re-usable software systems in order to ease the development process of software components. Frameworks are considered as such re-usable software assets.

A framework is defined as a set of classes that embodies an abstract design for solutions for a family of related problems, and support reuse at a larger granularity than classes [12]. They are defined as semi-complete applications which are specially designed for particular application domains. The infrastructure provided by the framework is then extended and specialized more to produce custom applications.

Object-oriented frameworks may be grouped in terms of different aspects. J. Gurp and J. Bosh (2001) identify these groups as: *application frameworks, support frameworks and domain frameworks* [9]. Application frameworks provide a compact design and implementation for services which may define a generic application. Support frameworks provide services to much specific support interfaces such as operating system input/output handling library or defining scheduling policies. They can be regarded as the starting point to develop an infrastructure for the programs.

On the other hand, domain-specific frameworks provide an extendible design and implementation for the domains they address. They aim to reduce the amount of work during the development of various applications of a particular domain. For example, a hardware-in-the-loop simulation infrastructure for tactical missile systems can be realized by a domain framework and with specialized models added, a particular application can be developed. Such frameworks generally contain more detail and specialized design since they target a narrowed problem set. This reduces the work which must be conducted to develop an application using such frameworks, since there is now more pre-implemented design and feature.

Object-oriented frameworks gained importance in software development due to their convenient characteristics. Firstly, they promote reusability and reinvention of core components for domain is avoided. They enable large-scale reuse by capturing an abstract design for system components and the core implementation for a domain [10]. Secondly, object-oriented frameworks enhance modularity by defining generic and stable interfaces and hiding core implementations details from the user [11]. Finally, they are favorable during the development of domain applications in terms of enabling to produce software applications that are flexible and maintainable. Applications developed using a framework often evolve easily since with the help of modular design new specifications and features can be added to the application with minimal effort.

2.2 Cooperative Simulation (Co-Simulation)

Recent cyber-physical systems compose of interacting heterogeneous components [14]. Due to this variety of models and components, these systems require complex integration and simulation techniques. Co-simulation is an integration and simulation technique which allows different simulation tools and models collaborate.

In a co-simulation environment, system is decomposed of different models from different modeling environments and these models run in a black-box manner on their environment. Each environment maintains its local state and has the

responsibility of calculation part. Intermediate results from these calculations of each simulation environment are exchanged.

Synchronization of simulation actors, i.e. various models from different environments, and coordination of data exchange between the actors are the responsibility of a *co-simulation engine* [14]. In a master-and-slaves architecture, such co-simulation engine is called *Master* and other simulation actors are called *Slave*. Master determines the execution order of co-simulation slaves, manages shared and intermediate variables, and coordinates data-exchange and event triggering.

At the start of each simulation time step, the master gathers intermediate results from each slave and sets the proper input values in order to prepare slaves for the next simulation step. Data exchange is restricted to this interval between two simulation steps: Slaves cannot communicate with each other directly. Slaves can communicate with only the master and they can perform data-exchange only at the specified communication points.

After data-exchanges are performed, the master sets duration for the simulation step. During this time interval, each slave executes their solver and performs calculation. When time is up for the simulation step, a data-exchange point starts and the master prepares the slaves for the next simulation step. Proper termination of slaves is coordinated by the master at the end of overall co-simulation.

Since models are simulated in their own modeling and simulation environment, co-simulation is a powerful technique to simulate systems that are composed of heterogeneous components [15].

2.3 Functional Mockup Interface (FMI)

This section describes the FMI standard in detail. Firstly, an overview for the specification and the history for the standardization are given. Secondly, the characteristics of an FMU are provided. Thirdly, FMI interface is described in terms

of co-simulation version. Finally, the computational flow of an FMU in the co-simulation environment is described.

2.3.1 Overview and History

FMI is a modeling standard which introduces a common tool-interface standard for dynamic system models. The specification aimed to interface management problem among the models which are developed by different tools and it is initiated by Daimler AG [13]. In automotive field, there exist many different vendors that supply subsystem models as well as many different modeling and simulation tools for specialized steps of system development processes. Hence, the problem of interface management of dynamic system models gathered the attention of the company and the idea of developing a standard interface is matured in one of the Information Technology for European Advancement 2 (ITEA2) project named MODELISAR. The highlights and evaluation of the project can be listed as below:

- MODELISAR was undertaken by many leading companies such as Volvo, DLR, Daimler, Dassault Systems, and AVL.
- The project started on July 2008 and finished on December 2011 with a new and open modeling standard “FMI”.
- The interface released for different usage of system models as model exchange and co-simulation. FMI for Model Exchange is released on January 2010 and FMI for Co-Simulation is released on October 2010.
- A version of FMI named FMI for Product Lifecycle Management (PLM) is released on March 2011. Main intention of this specification was to release a guide to generically handle all FMI related data needed in simulation.

After maintenance and further development of FMI specification is transferred to Modelica Association, on July 2014, FMI 2.0 for Model Exchange and Co-simulation was released at the same package with major extensions on model behavior compared to version 1.0.

2.3.2 Functional Mockup Unit (FMU)

An FMU is defined as a model that is generated to comply with the FMI standard. An FMU may conform to three different versions of the specification and can be in following forms:

- A co-simulation slave: Conforms to FMI Co-simulation
- The coupling part of a simulation tool: Conforms to FMI Model Exchange
- PLM model: Conforms to FMI for PLM

A PLM FMU provides interface to handle FMI related data in a product lifecycle management simulation environment [13]. The interface introduces services to handle edition, documentation, results management, post-processing, analysis and report of simulation data. It also includes a format description to communicate between PLM system and the authoring tools.

In model exchange, FMUs which are generated by a particular development tool are imported by another simulation tool and simulated by the solver of this target environment. In this way, system models of one environment are available by another modeling and simulation host. The model exchange FMU is defined by differential, algebraic and discrete equations with time, state, and step events [13]. The overall structure of model exchange FMU is given in Figure 1.

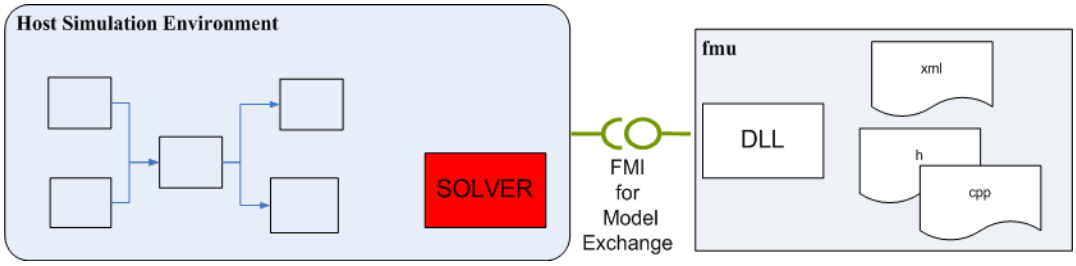

Figure 1 – Model exchange FMU and host simulation environment [13]

A solver is the supporting component of a simulation tool that solves complex continuous and discrete mathematical problems. Model exchange FMU pushes signals to the solver of the tool. Signals between an FMU model exchange and a solver is described in the specification as shown in Figure 2.

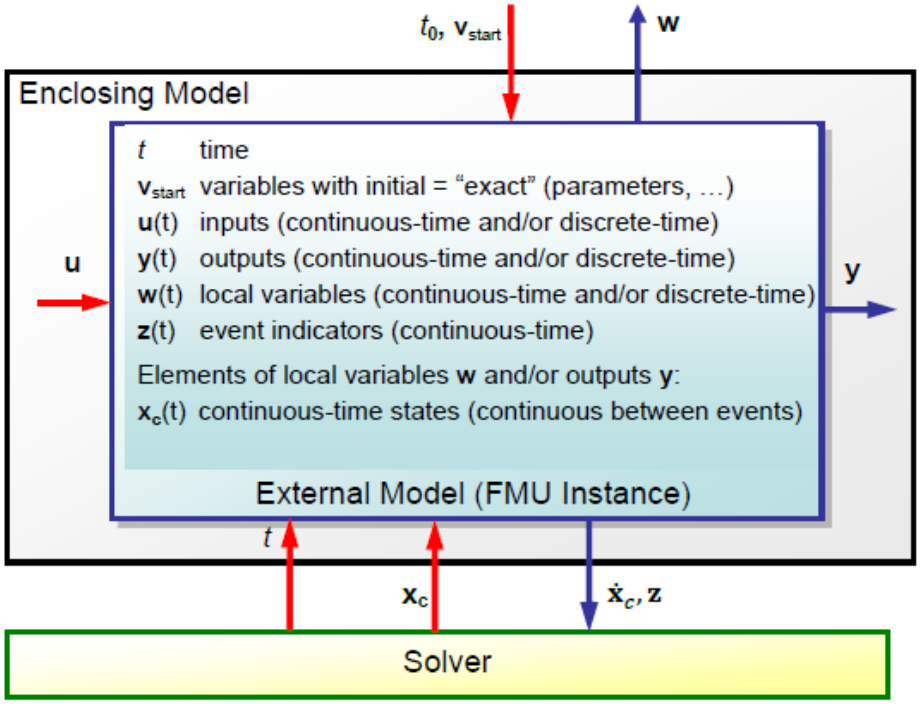

Figure 2 – Data flow between the environment and a model exchange FMU [13]

In co-simulation, coupling of various simulations tools are performed in a simulation environment. Co-simulation FMUs contain coupled sub-problem inside their solver and perform calculations between restricted communication points. General structure of a co-simulation FMU can be shown as Figure 3.

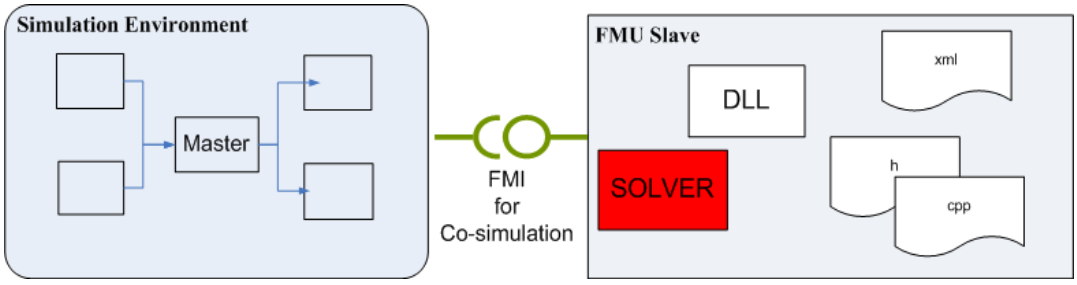

Figure 3 – Co-simulation FMU and a simulation environment [13]

Since a co-simulation FMU contains the solver, it performs as an independent component. The only requirement is the coordination with other simulation tools or

other co-simulation models. The coordination is done by the master model of the simulation.

Communication for coordination of models and tools can only be done on the restricted communication points. This situation is described in Figure 4, reproduced from the specification.

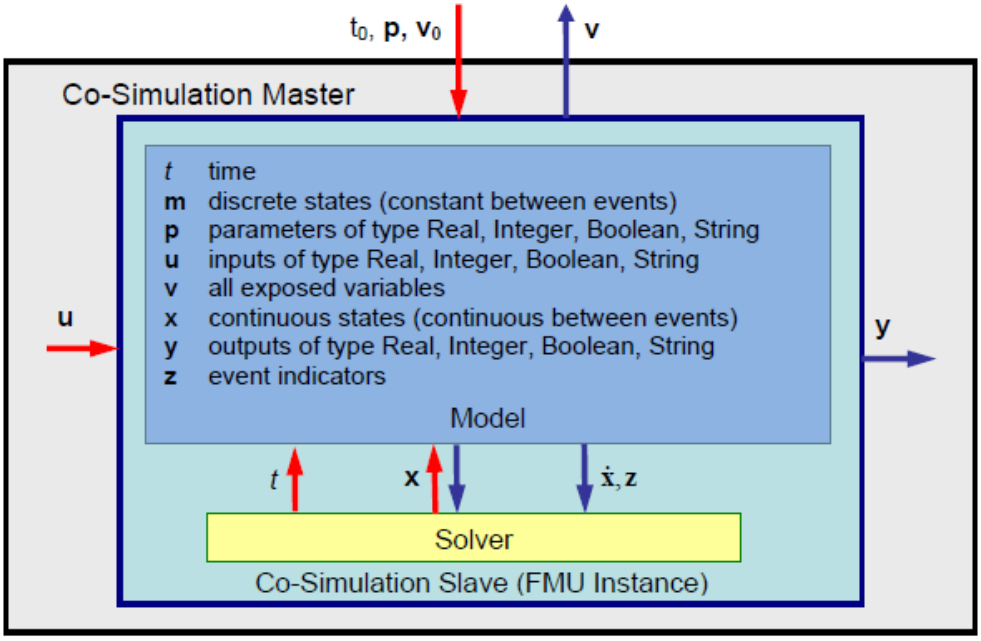

Figure 4 – Data flow between simulation master and co-simulation FMU [13]

FMU is distributed as a special archive file named as *ModelName.fmu*. The contents of this archive file are as follows:

- *modelDescription.xml (Required):* description of FMU
- *model.png (Optional):* image file of FMU
- *documentation folder (Optional):* FMU documentation folder
- *sources (Optional):* all C header and source files of FMU
- *binaries (Optional):* binaries in folders for particular operating systems and machine configurations

- *resources (Optional):* additional resources needed by FMU. This may be data which is read during initialization of the model or other files and folders.

The *.fmu* archive should either contain the sources or binaries folder.

The *modelDescription.xml* contains the model variables with their attributes such as name, unit, initial value and static information about the model. Structure of this xml refers to a schema called *fmiModelDescription.xsd*. The specification describes this schema as in Figure 5.

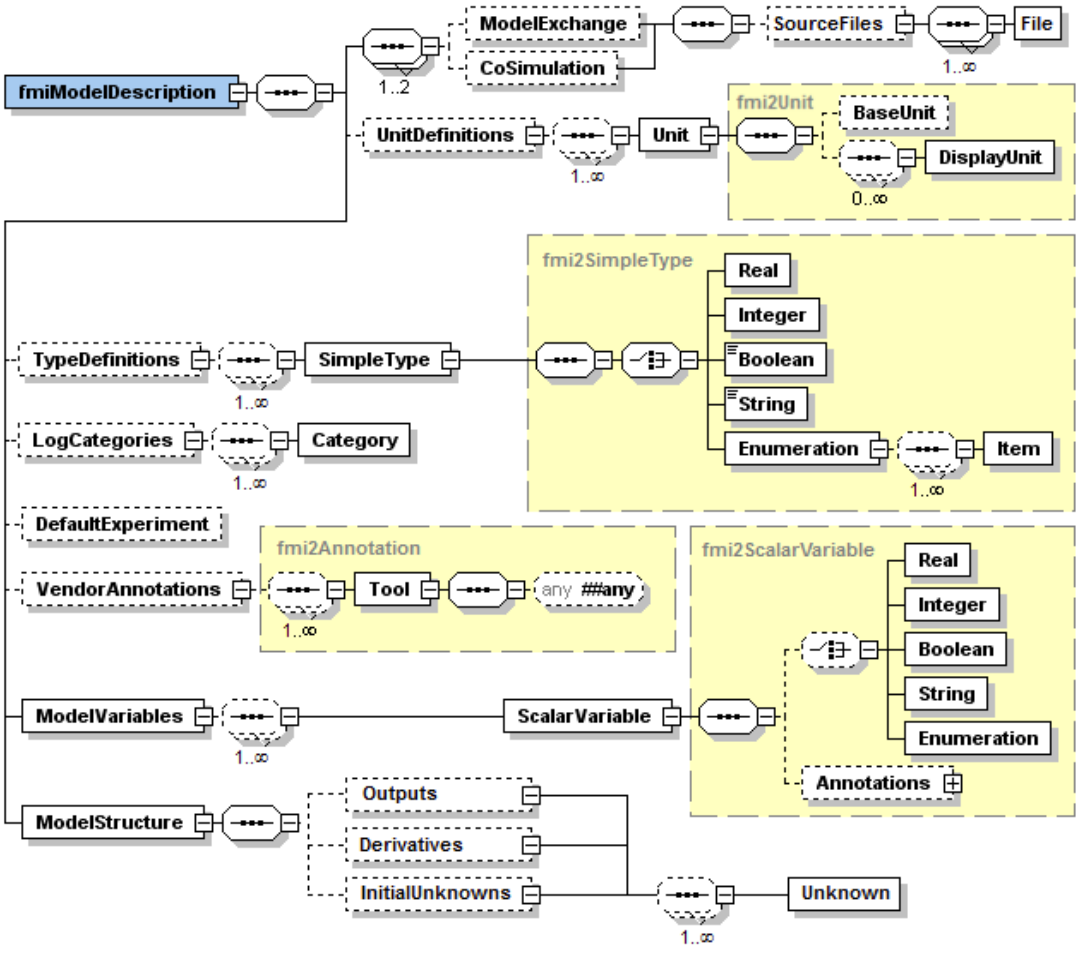

Figure 5 – Root-level FMU description schema [13]

2.3.3 FMI for Co-simulation

FMI for co-simulation is based on the master and slave architecture described in section 2.2 Cooperative Simulation (Co-Simulation). FMI specification provides an interface for the calculation of coupled subsystem problems independently for both time-discrete and time-continuous systems [13]. The data exchange can be done via master model only; there is no interface for slaves that enable to communicate directly to each other. Figure 6 shows basic master-slave structure of FMI.

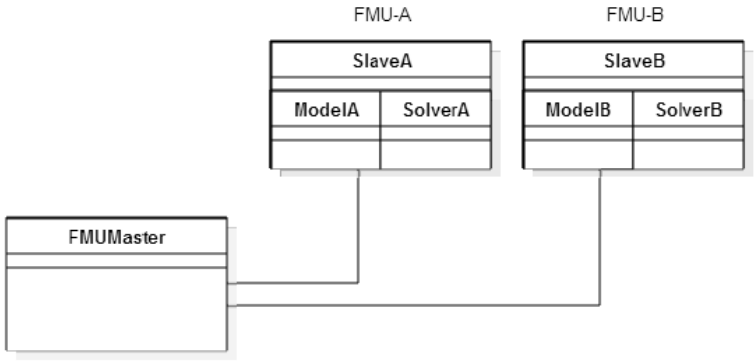

Figure 6 – FMI for co-simulation – master-slave architecture [13]

A minimal master algorithm stops the simulation of all slave models at each discrete communication point, collects the outputs, distributes the inputs to the slaves according to integration connections and triggers the slaves to continue with the calculations until the next communication time. FMI for Co-Simulation specification is designed to support generic master algorithms but it does not define the master algorithm itself [13].

FMU Co-simulation slave basically have following properties to support master coordination and perform computation accordingly:

- The ability to handle variable communication step sizes.
- The ability to repeat a rejected time step.
- The ability to provide derivatives as outputs for each step time.

- The slave can interrupt the master.
- Functions of solver needed to be time-dependent.

2.3.4 Computational Flow of FMU Co-simulation

In a simulation environment with master model and various co-simulation FMUs, steps of co-simulation can be seen in basically three phases, namely, Initialization, Computation and Termination as shown in Figure 7.

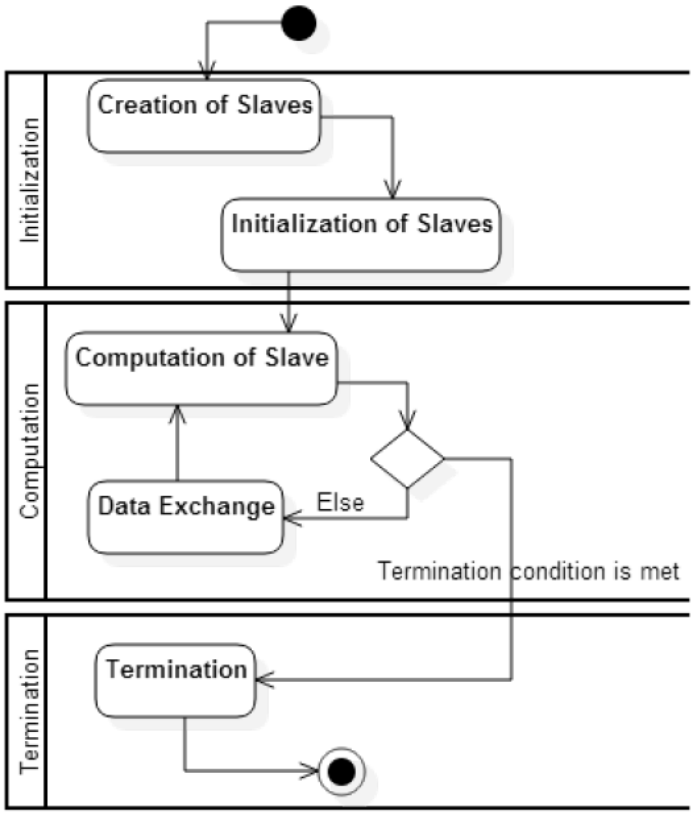

Figure 7 – Activity diagram showing basic steps in co-simulation of FMUs [24].

In the initialization stage, slaves are created and instantiated for the beginning of the simulation. Proper values of the static and dynamic variables of slaves should be set inside the instantiation part of FMUs. In the computation stage, all FMUs are triggered for performing their computation part, after they finish, termination conditions are checked to decide either to continue with next time computation or to

transfer to termination state. Finally in termination stage, termination parts of slaves are triggered and models properly close the simulation in order.

FMI for Co-simulation specification consists of methods to support these stages of co-simulation. Details of these methods and order of call for functions are given in the following section.

2.4 FMI for Co-simulation - Programming in C

FMI for Co-simulation specification covers all steps of co-simulation given in Figure 7. In addition to these steps of co-simulation, the FMI specification can be grouped in two as follows:

- Functions to perform the computation of coupled problem and synchronize the calculation step with the overall simulation
- Functions to share and receive intermediate results of computation

Considering these two classifications, FMI interface will be discussed in detail following the co-simulation steps given in Figure 7.

2.4.1 Initialization Interface

As discussed before, the first step for co-simulation of FMU slaves is creation of slaves and initialization of slaves. FMI specification does not include creation of an FMU. The specification begins with the initialization of FMU slaves. Table 1 summarizes the initialization functions [13].

Table 1 – FMI for co-simulation for initialization interface [13]

FMI Specification	Detail	Computational or I/O Operation
fmi2Instantiate	This function returns a new instance of an FMU. Initialization routine for the	Computational

	slave is performed at the beginning of the simulation.	
fmi2EnterInitializationMode	Triggers FMU slave to enter initialization mode.	Computational

2.4.2 Computation Interface

FMI specification has functions to perform computation and share intermediate results with FMU Master. These functions are:

- *fmi2DoStep*: Slave computation of a time step is triggered with this function
- *fmi2SetReal/Integer/Boolean/String:* Data exchange is controlled with these methods. The FMU Master sets and gets intermediate results for slaves.
- *fmi2GetXXXStatus*: These functions inform the master about simulation status of the slave.

Listing 1 shows a code snippet of FMU Master calling slave data exchange and computation interface.

```
//Simulation sub-phase
tc = startTime; //Current master time

while ((tc<stopTime) && (status == fmi2OK))
{
//retrieve outputs
s1_fmi2GetReal(s1, ..., 1, &y1);
s2_fmi2GetReal(s2, ..., 1, &y2);

//set inputs
s1_fmi2SetReal(s1, ..., 1, &y2);
s2_fmi2SetReal(s2, ..., 1, &y1);

//call slave s1 and check status
status = s1_fmi2DoStep(s1, tc, h, fmi2True);

switch (status) {

case fmi2Discard:
```

```
fmi2GetBooleanStatus(s1, fmi2Terminated, &boolVal);

if (boolVal == fmi2True)
printf("Slave s1 wants to terminate simulation.");

case fmi2Error:

case fmi2Fatal:
terminateSimulation = true;

break;
}
if (terminateSimulation)
break;
//call slave s2 and check status as above
status = s2_fmi2DoStep(s2, tc, h, fmi2True);
...
//increment master time
tc += h;
}
```

Listing 1 – Example computation state of FMU Master – data exchange and triggering slave computations [13]

2.4.3 Termination Interface

FMU Master controls the simulation time, termination conditions and status of the slaves and properly ends the simulation. To properly close slaves, FMI specification provides *fmi2Terminate* and *fmi2FreeInstance* functions. *fmi2Terminate* triggers termination of a slave and *fmi2FreeInstance* de-allocates slave instance.

```
if ((status != fmi2Error) && (status != fmi2Fatal))
{
        s1_fmi2Terminate(s1);
        s2_fmi2Terminate(s2);

        s1_fmi2FreeInstance(s1);
        s2_fmi2FreeInstance(s2);
}
```

Listing 2 – Example termination state of slaves [13]

2.4.4 Common Properties of the Interface

FMI specification aims to represent several instances with the same FMU and in addition to this, make it possible to have several instances in the same shared library of an FMU. This is achieved by defining all of the model interface functions as in the format of *modelName_fmi2xxx* [13]. This makes the usage of model interface harder since the name of the models should be tracked and erroneous calls should not be made by mixing the names of the model instances. Besides it demotes the code reuse since the definitions for the same functions are needed to be repeated.

Another common property of the model interact interface is that the signature of each function includes an argument that represents the FMU component. This argument is sent to the methods since the caller instance information such as GUID, name, FMI type is not known during the function call. The main reason for this situation is that the interface is provided in C, so there is no *object* concept.

Finally, most of the interface functions that appear in the computation step and are related to the variables of model instance, include value references in their signatures. Value references are definitions of indexes of variables in the value array of related data type. Since the values all of the model variables with the same type are kept in the same array, this information is needed in order to identify the variable. This problem can be eliminated with a pair list of the variable name and its value in a language like C++.

These common properties of the interface are guided the stage of mapping FMI to C++ API in the design of the MOKA Framework.

CHAPTER 3

RELATED WORKS

This chapter provides an overview of related works in FMU development and co-simulation. Firstly, tools that support the FMI standard are provided. Secondly, object-oriented approaches to FMI are reviewed. Finally, a brief comparison of these approaches with MOKA Framework is presented.

3.1 Tools that Support the FMI Standard

As the system development process proceeds, main focus becomes the interactions between models. Model integration and simulation are the main practices that take role for controlling and visualizing model interactions. In order to ease the model integration and use of different dynamic system models from different vendors in a simulation environment, a model interface standard FMI is defined as described in chapter 1, INTRODUCTION. FMI well received by the modeling and simulation industry and there are already tools to interact with FMUs.

The wide recognition of FMI modeling standard has led the existing well-established modeling and simulation tools to support the FMI standard by providing interfaces to import and export FMUs. The FMI Toolbox for MATLAB/Simulink® from Modelon [5] is an example to such tools for enabling FMU usage on MATLAB/Simulink® tool. Although the FMI Toolbox enables model developer to import already developed FMUs to the simulation environment on MATLAB/Simulink® tool and export models developed in MATLAB/Simulink® tool as FMUs, the toolbox does not provide any guidance to develop, integrate and co-simulate the FMUs from scratch.

At this point, in addition to the applications added to existing development tools, there are some libraries to interact with FMUs more than import/export. FMI Library (FMIL) by JModelica [6] and FMU Software Development Kit by QTronic [7] are most popular libraries for developing and simulation of FMUs.

Both of the applications are aimed to relieve the user from managing the details of FMU interaction and they are considered as a starting point for applications with interfacing FMUs. They provide a C API for interacting with aspects of the FMU they interact with including services for such as loading an FMU to the simulation environment and managing data flow among FMUs.

FMU SDK by QTronic aims the FMU construction guideline issue and introduces and FMU Template which is basically the implementation of the FMI methods. To develop a new FMU this template is used. This approach helps during FMU construction and may be considered as the starting point for enhancing reusability of components which are common in FMUs and hiding the details of implementation from the model developer.

3.2 Object-Oriented Approaches to FMI

In addition to such applications that provide a starting point on interacting with FMUs; there are also libraries that aim to extend FMI specification for object-oriented languages. The FMI++ Library [8] is a study that provides a utility package for interfacing with FMUs with a C++ API. The library consists of various class definitions for construction and simulation of FMUs. It extends the capabilities of FMU and provides services for generic advanced numerical integration and event handling capabilities. Since the API of the FMI++ library changes the interface specification of the FMUs instead of extending the FMI standard, this approach may not be favorable in cases of developing models which fit the standard.

There is another library to handle FMI specification with an object-oriented language that is Java FMI (JFMI) by University of Berkeley [18]. JFMI is a Java wrapper for FMI specification. The work is similar to FMI++ Library but it uses Java language.

3.3 Brief Comparison with MOKA

These libraries provide an insight for the development of modeling and simulation environments to interact with FMUs. The capabilities of these libraries and those of MOKA Framework are compared and listed in Table 2.

Table 2 – Comparison of the libraries that support FMI and MOKA Framework

Capability	**FMIL**	**QTronic SDK**	**JFMI**	**FMI++**	**MOKA Framework**
Object-Oriented Language Support	×	×	✓	✓	✓
FMI Standard Interface	✓	✓	×	×	✓
FMU Template	×	✓	×	✓	✓
FMU Master Template	×	×	×	×	✓
Support for Co-simulation Algorithms	×	✓	✓	✓	✓
Import FMUs	✓	✓	✓	✓	✓
Model Exchange/ Co-simulation	Both	Both	Both	Model Exchange	Co-simulation

To sum up, FMI modeling standard is widely recognized by the modeling and simulation industry and there is a tendency towards developing tools to interact with FMUs in various modeling and simulation environments. At this point, there are already some popular tools and libraries to interact with FMUs, but these approaches are needed to be extended in terms of the following:

- They need to offer more capabilities than importing/exporting of FMUs.

- The model developer needs guidelines for FMU development as well as FMU master development.
- Well-established software development approaches like object-oriented programming need to be supported.

MOKA Framework aims to combine these capabilities needed by the model development and co-simulation processes with FMUs.

CHAPTER 4

MOKA FRAMEWORK

This chapter introduces the MOKA Framework. First of all, an overview for the MOKA Framework is provided and main characteristics are pointed out. After this sub-section, the methodology that is used in MOKA Framework for mapping the FMI specification to an environment of C programming language to a programming environment with C++ language is described. After that, the logical architecture of MOKA Framework is presented and class details are provided. Finally, design rationale for the MOKA Framework is provided.

4.1 Framework Overview

FMI provides a standard interface for dynamic system models to fit, but the specification does not include the realization of the interface it introduces. In order to develop models using FMI specification, model developers need a guideline to realize methods in the interface.

As the model and system development process proceeds, main focus moves to subsystems and relations between them. At this point, subsystems mature with the development of dynamic models and with the integration of these models, the overall system takes form. The integration process of the models is delineated more clearly with the help of FMI standard, but the developers still need a compact infrastructure to ease the process. Additionally, code and component reuse should be promoted in such systems. This can be achieved with a modular approach. But, since FMI standard is intended to focus on platform independency, high level software development approaches like object-oriented methodology is left-out from the specification. In order to promote modularity and component reuse more effectively, FMI standard should meet with object-oriented development approaches.

Combining these needs for develop, integrate and co-simulate dynamic system models, MOKA framework aims to provide an infrastructure for:

- Model generation – by introducing an FMU template
- Model integration – by introducing an FMU Master template
- Running complex co-simulation scenarios

By providing such infrastructure, MOKA framework contributes to model based system development process with FMI as it:

- Guides the model developers on constructing models with FMI compatibility.
- Addresses the need for an infrastructure that supports a simulation environment with FMI and provides a model generation, integration and usage environment for co-simulation in a unique, complete framework that supports FMI standard.
- Fills the gap between basic FMI specifications and typical co-simulation integration needs by extending the specification with an object oriented modular structure.

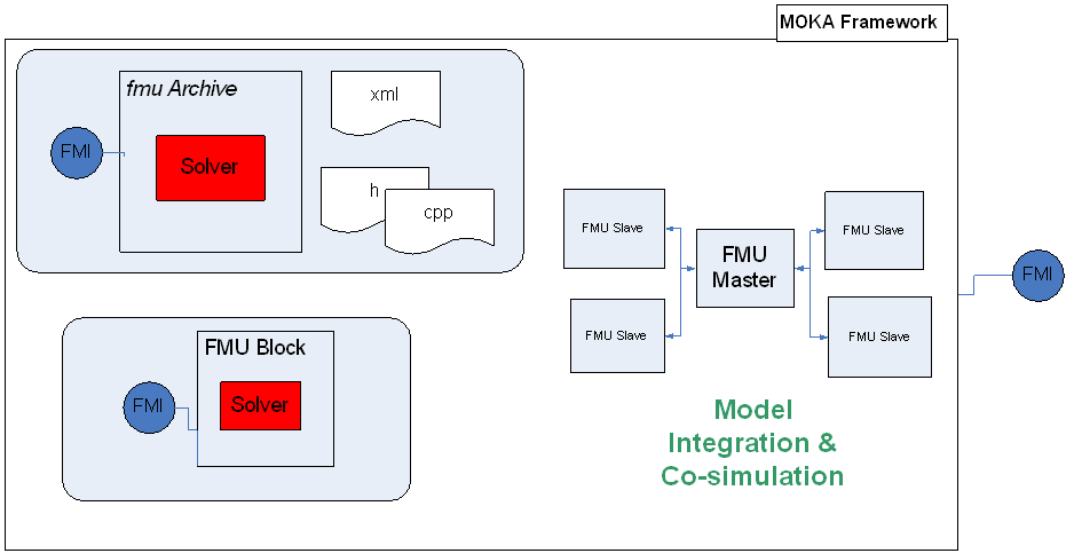

Figure 8 – Description of MOKA Framework

Figure 8 introduces the design of MOKA Framework which satisfies the listed requirements and contributions. The environment of MOKA Framework consists of an FMU Master and co-simulations models. Co-simulation model pool is heterogeneous in terms of development tool. In other words, models which will be integrated and simulated in MOKA Framework, can either be developed by using MOKA Framework itself or they can be generated from another modeling and simulation environment in the *.fmu* form. Models generated by other modeling tools are used with a shared library inside the MOKA Framework.

4.2 Mapping FMI to C++ API

This section describes the implementation details of the MOKA Framework from the perspective of the programming interface of FMI specification. The description covers how the FMI specification is mapped from an environment of C programming language to a programming environment with C++ language.

In order to create an object-oriented framework for co-simulation of FMU slaves, the programming language should be changed from C, a procedural language, to an object-oriented language. The MOKA Framework is designed in a C++ environment for such reasons: C++ language is commonly used object-oriented programming language for system development, modeling and simulation, and it is an already proven language in terms of overheads of object-orientation and memory management. The C++ language provides powerful realization of object-orientation with such tenets as inheritance, polymorphism and encapsulation. The C++ language is recognized as the one of the most fast, powerful and flexible programming languages for these and more features [19] [20] [21].

The main focus of the programming language mapping process was to keep the model usage interface strictly the same as the FMI specification. This way, the standard is not compromised. Models developed with MOKA Framework then have the potential to be used in other simulation environments with standard FMUs. Additionally, since the interface of the usage of an FMU is standard, models that are

developed in other modeling and simulation environments can be used in MOKA Framework together with the framework models.

The interface of model interactions remains the same, but the underlying data structures are re-designed. The data structure definitions are mapped to C++ compatible data types. To illustrate, variables of an FMU are needed to be kept in a static array collection. So, the index of the variable in the array of the corresponding variable type is needed to be known. This situation makes it harder to find a variable in the case where there are a lot of variables with the same type as described in section 2.4.4 Common Properties of the Interface. But in C++, the variables list is kept in a dynamic map, which eliminates the responsibility of the programmer to know the index of each variable.

Since a primary concern is to keep the model interface same as the FMI specification, some of the arguments in the function signatures are kept even though they are not used. To illustrate, most of the functions defined in the FMI specification takes an *fmuComponent* argument that reflects the FMU model instance. The caller model is identified with this argument. This is not needed in C++, since now the instances are represented with objects and the functions are defined as member functions.

4.3 Logical Architecture

This section presents the logical architecture of MOKA Framework and introduces its object-oriented structure which is designed to generate, integrate and co-simulate FMU slaves promoting modularity and code reuse. Design rationale of MOKA Framework is described in the section 4.4.

Considering the advantages of block-based development on model development and integration that are described in section 4.4 Design Rationale, MOKA Framework development adopted block-and-port-based structure. An FMU can be abstracted to contain components for data exchange and computation for the co-simulation. The data exchange component of a FMU is abstracted as a port and the computation

component of a FMU is abstracted as a block. The UML class diagram in Figure 9 shows the logical view of the MOKA Framework and demonstrates these abstractions.

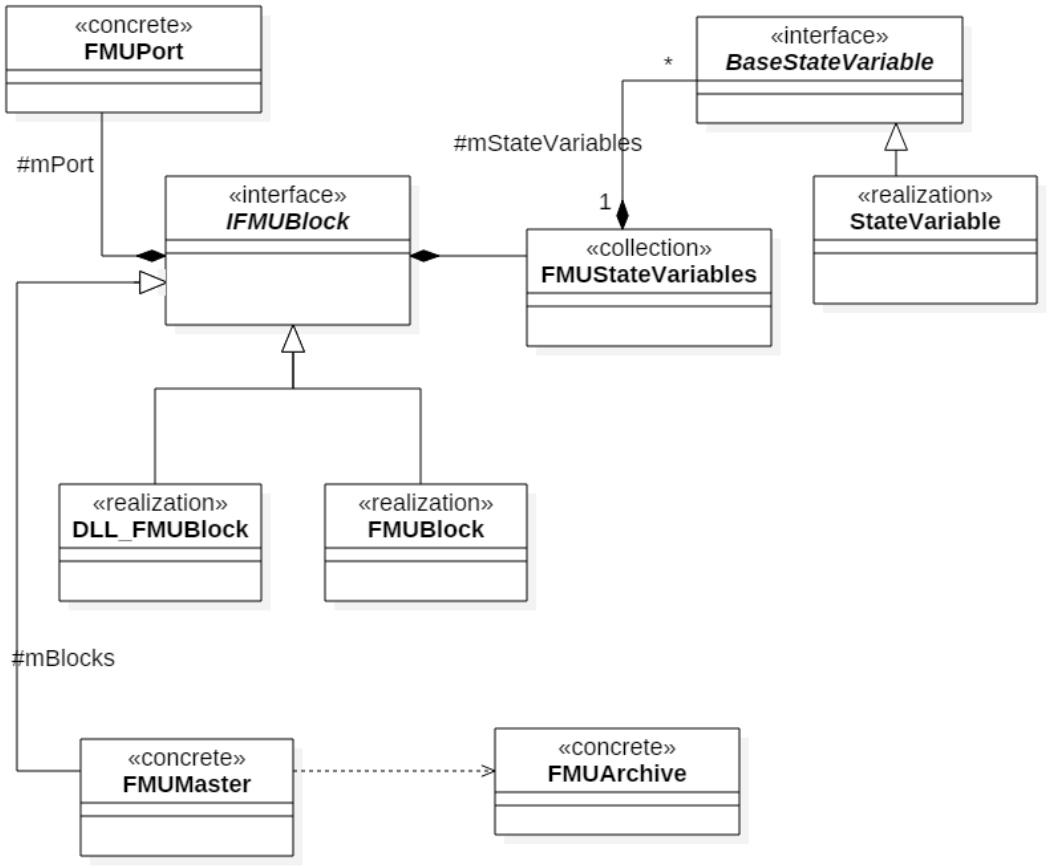

Figure 9 – Class diagram of MOKA Framework – top view

FMUPort class represents the port abstraction of FMU while the IFMUBlock represents the block abstraction. FMUBlock is the realization of this abstract block component for the models which will be developed using MOKA Framework. DLL_FMUBlock is the realization of IFMUBlock for the models that will be imported to the framework from other modeling tools. The computation part is further divided to separate state tracking operations of the model. State variable of a model is abstracted under BaseStateVariable class and realized in StateVariable

class. Since an FMU can contain more than one state variable, to hold these variables inside one collection FMUStateVariables class is defined.

Details of each class are presented in following sub-sections of this heading.

4.3.1 IFMUBlock

The first step to represent a model object was to define the structure of model block to support availability of heterogeneous models in the framework. IFMUBlock is intended to represent both models developed in MOKA Framework and models generated using other modeling tools that support export to FMI.

This class is designed as abstract and provides interface for interacting with FMUs during the simulation. FMUMaster uses both models developed in MOKA Framework and models imported from other modeling tools as FMU with the same interface provided by IFMUBlock.

The class contains the function definitions for the computational part of FMI Co-simulation version 2.0. The signatures of these functions are not changed in order to preserve the standard interface. The list of function definitions for the computational part of FMI Co-simulation version 2.0 that are included in IFMUBlock class can be seen in the Appendix A - FMI CO-SIMULATION INTERFACE FUNCTIONS.

4.3.2 FMUBlock

FMUBlock realizes the IFMUBlock class for the models which will be developed using MOKA Framework itself. The class represents the model template and provides an abstract FMU structure of the MOKA Framework. A concrete FMU is developed by extending this class. In order to ease FMU construction process, this class implements the FMI computation interface defined in IFMUBlock. For the functions whose implementation varies model to model, such as initialization of beginning inputs, the generic implementation achieved by providing pure virtual operations for them.

Figure 10 shows the reduced class definition of FMUBlock class. The FMI 2.0 computation part function definitions that are realized from *IFMUBlock* class are not included in this view.

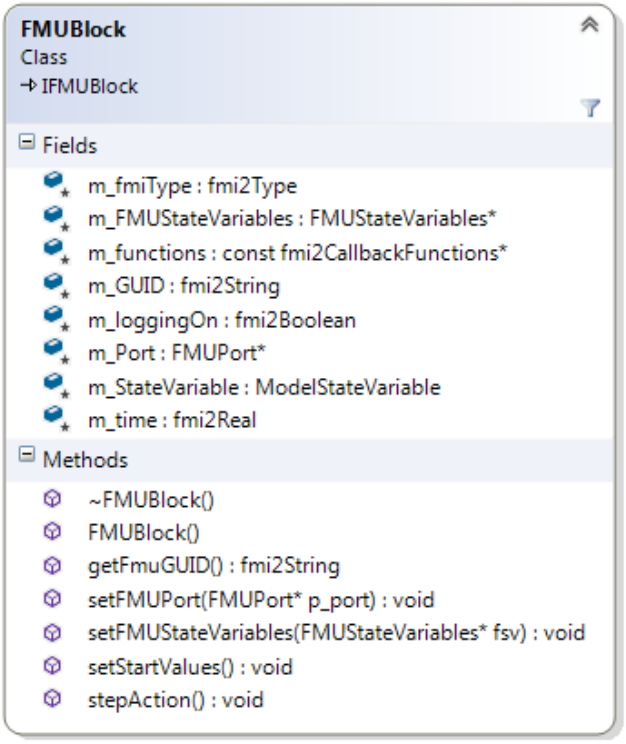

Figure 10 – FMUBlock class details

FMUBlock class guides the FMU developers for the development of a model that conforms to FMI standard. Additionally, the class promotes code reuse by providing FMI 2.0 co-simulation interface implementation and it proposes a modular structure by consisting of FMUPort and FMUStateVariables. A suitable FMUPort that is implemented and ready to use can be connected to the FMUBlock and the input/output interface management can be achieved.

4.3.3 DLL_FMUBlock

DLL_FMUBlock is designed in order to enable integration and co-simulation of FMUs which are developed by using other modeling environments. FMUs which are developed using another modeling and simulation tool and packaged as *.fmu* can join

the co-simulation environment in MOKA Framework seamlessly with the help of this class.

DLL_FMUBlock realizes IFMUBlock class and implements the FMI interface functions by calling directly to corresponding shared library methods of imported FMU.

4.3.4 FMUPort

Simulation data and data exchange services of an FMU are encapsulated in FMUPort class. The class contains data vectors that represent simulation variables of the developed FMUBlock instance.

In the FMI 2.0 standard simulation variables of the FMU is kept in a static array and there is no interface to access a specific variable through its name. The variables are accessed with their index in the array. These index definitions are called *value references*. Maintenance of the value references is the responsibility of the FMU developer in the standard. With the help of dynamic C++ collectors, FMUPort class eliminates the need to manage the value references. The value reference of a variable is automatically assigned and kept inside the FMUPort class and when an FMI function is needed to be called, the value reference is provided to the user via a function call.

The concrete master creates FMUPort objects and connects them to the corresponding FMUBlocks.

4.3.5 BaseStateVariable

During the simulation, the computation algorithm of a dynamic model updates the values of pre-defined variables of the model. The values of such dynamic variables describe the state of a model and represent the outputs of computational part. BaseStateVariable class is designed to stand for such abstract state variable component for an FMUBlock. The class provides minimum interface to define state variables and defines virtual functions to initialize and update the value of the state

variable considering the simulation time. The details of signatures of these methods and data members can be observed in Figure 11.

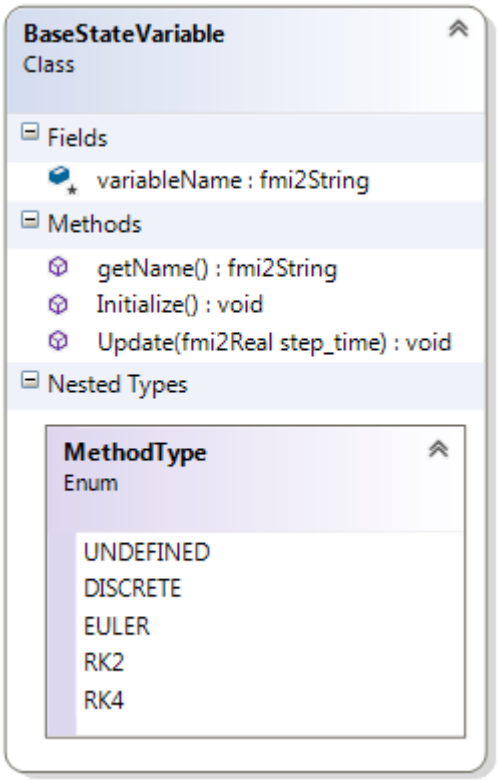

Figure 11 – BaseStateVariable class details

A co-simulation slave contains its solver inside as described in section 2.3.2 Functional Mockup Unit (FMU). A state variable changes its value in computation steps by the nature of the solver. BaseStateVariable class provides an enumeration type for the integration method of the FMU solver. Hence, the concrete StateVariable class can implement the integration method and update the value of state variable accordingly. This enumeration can be extended to define other solvers.

4.3.6 StateVariable

StateVariable class realizes the abstract BaseStateVariable class and represents the state variable object of the FMUBlock. The class designed as template in order to achieve a general representation for different types of variables.

StateVariable class has data members to keep track of initial state value, current state value and previous state value in order to keep track of the state history of the FMU slave. Since continuous dynamic models do computation with derivatives, the derivatives of the current and previous state values are also stored in data members. The class has member functions to access initial state value and gather current and previous state value as well as the derivatives. Figure 12 shows the class details for StateVariable class.

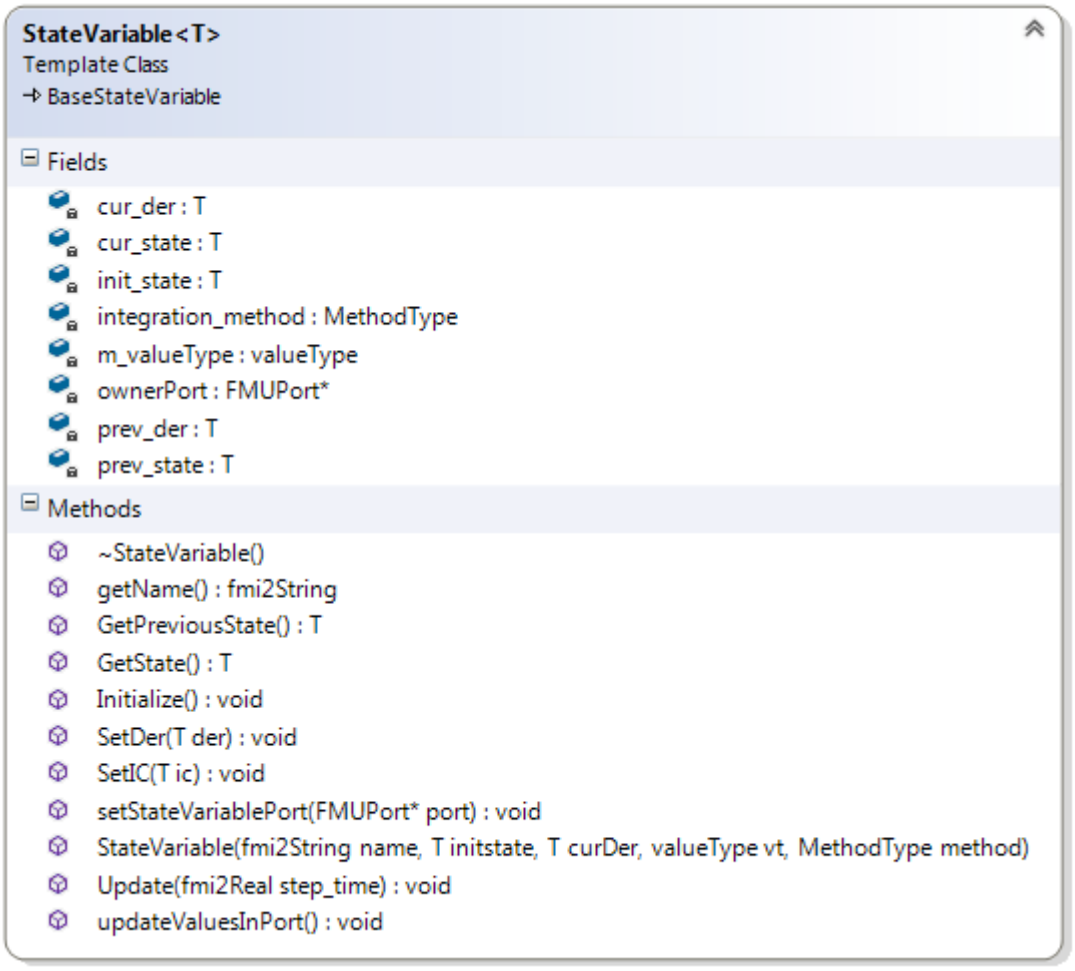

Figure 12 – StateVariable class details

During a computation step, the values of the state variable and its derivative are updated by *Update (fmi2Real steptime)* function according to assigned integration method. The enumeration defined in BaseStateVariable class and assigned as data

member in StateVariable class is checked in the *Update (fmi2Real steptime)* function and proper calculations for the next state and its derivative is assigned. Listing 3 gives an example calculation for the *EULER* integration method.

```
switch (integration_method) {

      case BaseStateVariable::EULER:
            {
                  prev_state = cur_state;
                  cur_state += cur_der*step_time;
            }
            break;
      (…)
}
```

Listing 3 – Calculation of state and its derivative according to integration method

After the value of the current state and its derivative is calculated, the value of the variable should be updated in the FMUPort of the FMUBlock. In order to handle this, StateVariable class holds a data member which points to the owner FMUPort. Proper assignment of the owner port is done during the initialization of FMUBlock by the master FMU.

4.3.7 FMUStateVariables

FMUStateVariables class represents the state variable list of an FMU slave. The class provides services for the operations on this list such as adding a new state variable to the list, or gathering the value of a specific state variable. Figure 13 shows the details of FMUStateVariables class.

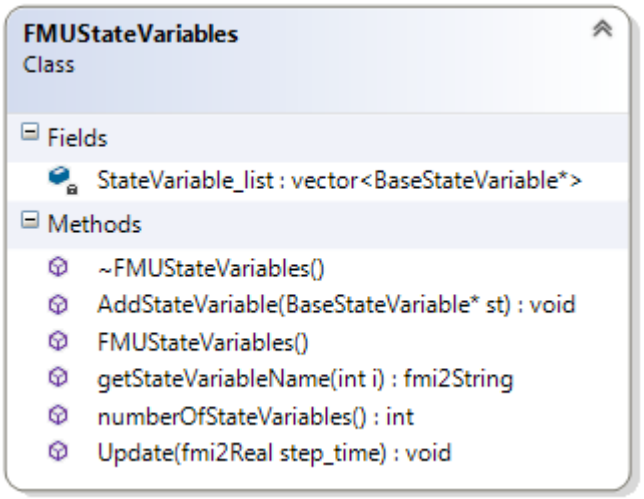

Figure 13 – FMUStateVariables class details

FMUBlock class has FMUStateVariables and during the computation *Update (fmi2Real steptime)* method is called. The method then triggers *Update (fmi2Real steptime)* functions for all state variables in the list. This way, in pursuit of the completion of a computation step, all of the state variables change their value according to their own assigned solver.

4.3.8 FMUArchive

MOKA Framework allows FMU slaves that are generated from another modeling tool to join the simulation environment of MOKA Framework. Such FMUs must be pre-processed before they are ready for use in co-simulation. FMUArchive class is designed to coordinate such pre-processing operations.

FMUArchive class has the responsibilities of providing services to import the *.fmu* shared library of the foreign FMUs. The class has methods to unzip the *.fmu* package, parse the *modelDescription.xml* and loading the shared library of the FMU slave. To unzip the *.fmu* package, the 7-zip [22] open source file archiver is used. With the extraction of *.fmu* package, the directory structure described in section 2.3.2 Functional Mockup Unit (FMU) is obtained. For xml parsing, the *tiny xml-2* [23] C++ xml parser is used.

After the extraction of *.fmu* package, shared library of the FMU slave is loaded. FMUArchive creates an empty DLL_FMUBlock instance and the handle to this shared library is connected to this instance. FMUArchive returns the imported block to the concrete FMUMaster in order to use in simulation.

4.3.9 FMUMaster

MOKA Framework introduces a template FMU master for the integration and co-simulation of slaves. The design of the FMUMaster aimed to promote modularity and compactness in slave design. At this point, a slave visualized as a compact system that might consists of several FMU slaves and a master that coordinates these slaves. In order to integrate and co-simulate such compact slaves in the simulation environment seamlessly, the idea of designing the master interface as to obey FMI specification is useful.

To achieve this, FMUMaster class inherits from FMUBlock and gathers FMI interface. This way already integrated models can be used as a slave in a simulation environment provided by MOKA Framework.

The responsibilities of FMUMaster class can be listed as follows:

- The FMUMaster creates FMUPorts that will be connected to the FMUBlocks and initializes them.
- The FMUMaster creates and initializes FMUStateVariables for each corresponding FMUBlock.
- The FMUMaster creates the FMUBlocks and sets the state variables and ports of the slave.
- The FMUMaster loads the FMUs that are generated by other simulation environments.
- The FMUMaster initializes all FMU slaves and makes them ready for the computation state. After being ready for the co-simulation, the master calls

fmi2DoStep function for all slaves in order to trigger solvers of the slaves for separate computation.

- At the end of each computation state, the master gets outputs from each slave and sets inputs to each slave. For all computation steps, the termination condition is checked by the master. If the termination condition is met, the master should manage proper de-allocation of memory and termination of slaves.

Implementing these responsibilities, FMUMaster has a design as described in Figure 14.

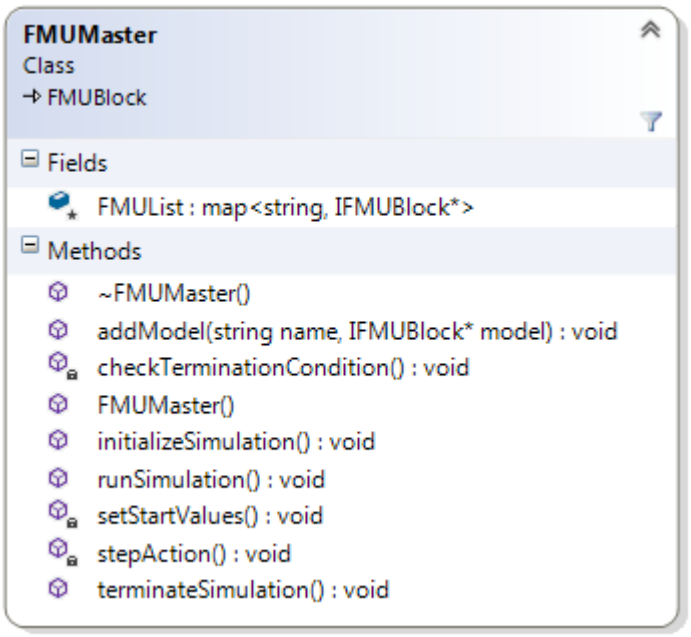

Figure 14 – FMUMaster class details

4.4 Design Rationale

This section introduces the design rationale of MOKA Framework. The top-view logical architecture design decisions as well as the considerations when designing each class are described.

Complex cyber-physical systems consist of a variety of components which interact with one another according to the nature of the represented system [29]. There are already well established and widely accepted formalisms to represent cyber-physical systems on a component-based approach. Discrete Event System Specification (DEVS) [33] and Systems Modeling Language (SysML) [34] are two well-known examples of such formalisms. Both formalisms define models in a composite manner and separate input/output mechanisms of the model from the model computation. Additionally, recent model development tools promote the component reuse, semi-automatic code generation and more clear subsystem design in order to develop faster models. At this point of view, industry-wide well-employed modeling tools such as MATLAB/Simulink® [16], and Scilab/Xcos [17] describe models in terms of functional components called blocks and data connection units called ports. A block can be described as a functional component or model which consists of a solver and some specialized virtual input and output ports. A port can be described as a component which holds data and provides data exchange services.

Several specifications of block-port based model development approach promote a well visualized, faster and managed system design for model based development [30]. One of these specifications is that ports can be developed independently from the block and provide specialized data types and data exchange services compatible with the domain of interfaced blocks. As the result of afore-mentioned constraints on the ports, block input and output interfaces can be defined to be connected with only special set/sets of ports. This way semi-automatic input and output flow control can be achieved. The other specification is that model integration process can be reduced to only setting proper port connections throughout the blocks. Output port of a block is directly connected to input port of another block similar to MATLAB/Simulink®. In the phase of integration, coverage of the data interfaces is considered without paying attention to the other parts of model. Hence, achieving the correct integration between components means to provide a meaningful connection between the input and output ports of each block in the system [29]. Finally, block-port based model development enables cascaded structure and promotes modularity. A block can

consist of any number of blocks integrated inside. This way by chancing the combinations and integration mode of the sub-blocks, various master blocks can be developed with the help of this modular and cascaded structure. Hence, many types of applications can be developed using the modular and hierarchical architecture that is provided by the block-based organization [31].

Since the block-based architecture provides such advantages, MOKA Framework has adopted the block-based approach. In order to represent a model component as a block, IFMUBlock class is designed as described in the sub-section 4.3.1. MOKA Framework aimed to combine FMUs that is developed by using the framework itself and FMUs that come from other modeling tools. This requirement caused a design challenge for the representation of an FMU block since it is aimed to provide a model template by implementing the FMI specification. On the other hand, the imported models that are built by using other modeling tools implement the specification with the settings of the owner modeling environment. The models packaged as *.fmu* and reached as a shared library. Thus, the representation of an FMU block is designed as the standard interface of FMI co-simulation version 2.0, and the upper layer of this representation is divided into two as DLL_FMUBlock to hide the shared library usage inside the FMI interface implementation to import models from other modeling environments and FMUBlock to provide to actually implement the FMI interface. This way, with the help of two-layered design, FMUMaster considers two different model types as with a common interface and does not need to know the exact type of an FMU block.

Interconnections of IFMUBlocks are represented by FMUPort class that is a collection of simulation variables and services that serve communication with the FMUMaster. This class represents the model port in the block-based model development methodology. With the help of FMUPort class, the data part of an FMU block can be extended considering the requirements of the specific simulation setting. To illustrate, input-output requirement and synchronization of an FMU block can be controlled by the connected FMUPort instance. Hence, various simulation

configurations can be achieved by changing the connected FMUPort of an FMU block.

State variables of an FMU Block are encapsulated in BaseStateVariable, StateVariable and FMUStateVariables classes. BaseStateVariable is designed as abstract in order to have a common representation for various types of state variables. This design enables the framework user to enhance the capabilities of the state variable implementation provided by StateVariable class inside the MOKA Framework. The user may develop another realization of BaseStateVariable with different capabilities and use this implementation without interfering with the user classes of state variables. FMUStateVariables are designed as a collection class for the BaseStateVariable instances in order to provide an interface to perform a common task for all of the state variables of an FMU block in one shot.

The coordination of co-simulation and integration of models are the responsibilities of FMUMaster class. FMUMaster class is designed in such a way that it inherits from IFMUBlock class. This means that the class contains the FMI specification interface and it is also represented as an FMU block. This design supports hierarchical structure for the components that contain blocks inside blocks. A minimal FMUMaster instance coordinates the blocks inside a component and the component itself can be exported as an FMU block with the help this minimal master. This master model then can be controlled by another one-level-up FMUMaster. This design represents the master model as a block, promotes modularity and hierarchical component construction.

CHAPTER 5

MOKA FRAMEWORK USAGE

This chapter describes the usage of the MOKA Framework in three cases: constructing an FMU slave, constructing an FMU Master to coordinate the simulation and performing co-simulation of slaves.

5.1 Constructing an FMU

In this sub-section development of a new FMU using MOKA Framework is described. The steps of development process are illustrated using a toy example called BouncingBall. BouncingBall FMU is a simple ball model that simulates bouncing. The model is taken from Qtronic FMU SDK sample models [7].

The construction of an FMU using MOKA Framework can be described in three parts as: creation of a concrete FMUBlock that represents the FMU, creation of FMUPort for input/output handling and creation of FMUStateVariables for model state handling. The process and sub-steps are shown in Figure 15.

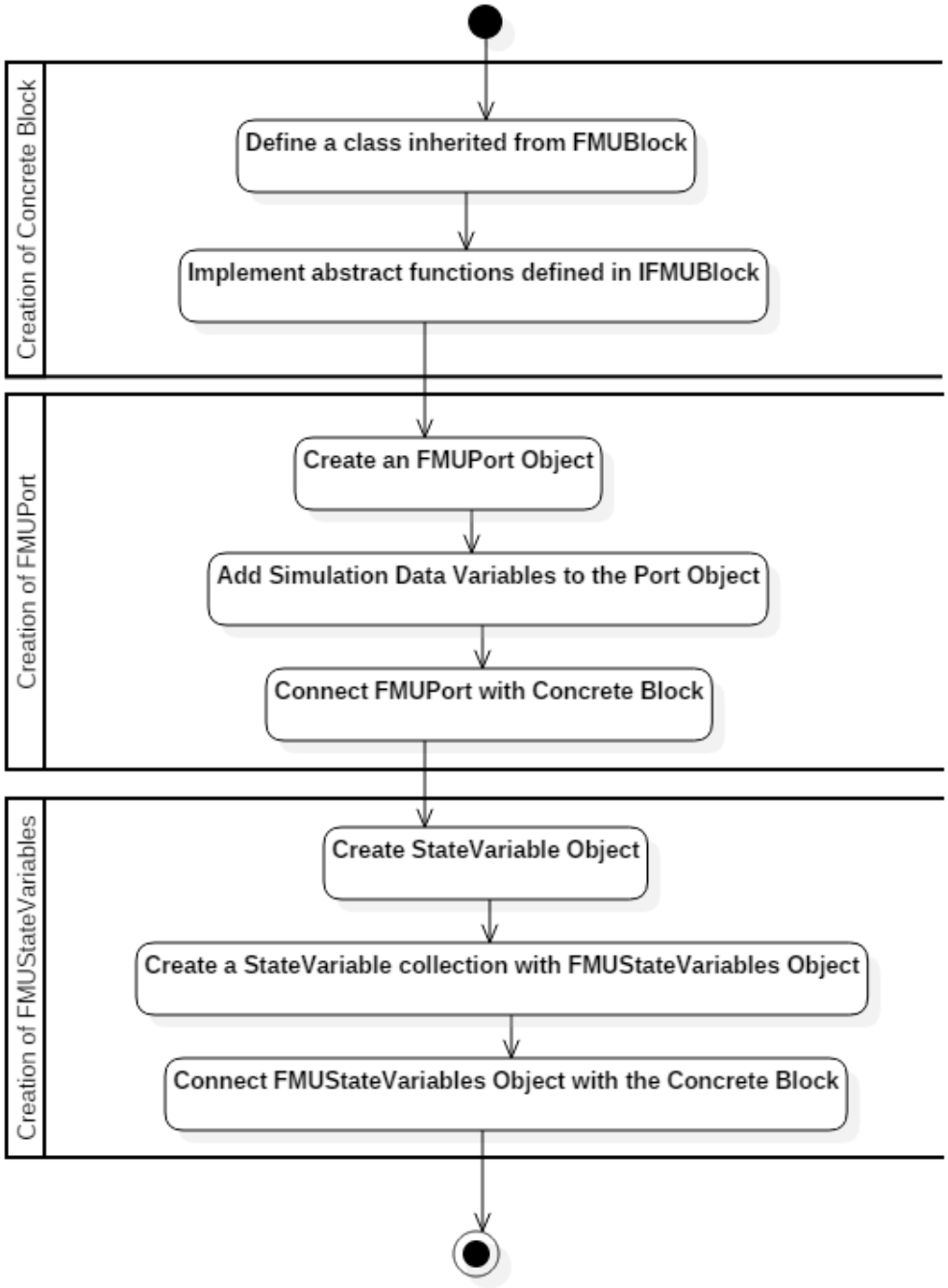

Figure 15 – Activity diagram for FMU development with MOKA Framework

The first step is creation of a concrete FMUBlock that represent the real FMU slave. In order to achieve this, a class that inherits from FMUBlock class is needed to be defined to represent the real FMU slave.

Since the methods whose implementation does not change from model to model are already implemented in FMUBlock class, the only need is to implement virtual functions `setStartValues` and `stepAction`. Listing 4 demonstrates the creation of a concrete class that will be represent the BouncingBall FMU slave.

```
// BouncingBall FMU
class BouncingBall : public FMUBlock
{
      public:
            // Constructor
          BouncingBall(fmi2String     pInstanceName,
                              fmi2String pGuid);

            // Destructor
            ~BouncingBall();

            // called by fmi2Instantiate
            virtual void setStartValues();

            // called by fmi2DoStep
            virtual void stepAction();

      private:
       (...)
};
```

Listing 4 – Creating a concrete class to represent BouncingBall FMU

In order to complete the development of an FMU using MOKA Framework, components that construct an FMU should be created and then they should be connected to the FMU concrete object. These components that should be created and initialized accordingly are FMUStateVariables and FMUPort objects. The FMUStateVariables object contains StateVariables in a collection.

First of all, an FMUPort instance should be created and input/output variables of the slaves should be added to this port. Listing 5 shows the code snippet for the example construction of FMUPort and input/output variables of BouncingBall FMU.

```
// Create an FMUPort instance

FMUPort* bouncingBallPort = FMUPort();

// Add input & output variables of the slave to the Port

bouncingBallPort->addRealVariable("h", 1.0);

bouncingBallPort->addRealVariable("der_h", 0.0);

bouncingBallPort->addRealVariable("der_v", 0.0);

bouncingBallPort->addRealVariable("g", 9.81);

bouncingBallPort->addRealVariable("e", 0.7);


// Set BouncingBall port

bouncingBall->setFMUPort(bouncingBallPort);
```

Listing 5 – Creating and initializing FMUPort for BouncingBall FMU

After creating and initalizing the input/output port of the model, state variables should be determined and added to an FMUStateVariables instance. This instance then connected to the BouncingBall model. Only variables that define the state of the model, i.e. variables that change their value during the simulation according to the defined solver, are instantiated. Listing 6 demonstrates the creation of state variables and connection of FMUStateVariables instance to the BouncingBall FMU slave.

```
// FMUStateVariables instance
FMUStateVariables* fmuStateVariables = new
FMUStateVariables();

// 1st argument is name,
// 2nd initial state value,
// 3rd initial derivative,
// 4th is valueType,
```

```
// 5th is default EULER solver
StateVariable<fmi2Real>* hState = new
StateVariable<fmi2Real>     ("h", 1.0, 0.0, RealType);

StateVariable<fmi2Real>* vState = new
StateVariable<fmi2Real> ("v", 0.0, 0.0, RealType);

// Set state port
hState->setStatePort(bouncingBallPort);

vState->setStatePort(bouncingBallPort);



// add state variables to the list
fmuStateVariables->AddState(hState);

fmuStateVariables->AddState(vState);

// Connect FMUStateVariables list to block
bouncingBall->setFMUStateVariables(fmuStateVariables);
```

Listing 6 – Creating and initializing state variables of BouncingBall FMU

5.2 Loading an FMU

This sub-section describes loading of an FMU that is developed by using another modeling and simulation environment.

The first step of the import process of an FMU to the MOKA Framework is to make the *.fmu* package available in the proper folder location. The default location for the *.fmu* packages of the models are the *fmus* folder located under the MOKA Framework structure.

The second step is to create an IFMUBlock instance and triggering the FMU loading by FMUArchive class. Throughout the simulation, FMUMaster knows only the common interface IFMUBlock for the simulation slaves. Hence, an FMU that will be imported should be mapped to the IFMUBlock interface. The shared library operations are encapsulated in the concrete class of IFMUBlock that is the DLL_FMUBlock class as described in section 4.3.3. In order to achieve this

configuration for the FMU that will be imported, FMUArchive class should be triggered to export the *.fmu* package and to load the DLL.

Figure 16 shows the steps of the importing process of an FMU that is developed using another modeling and simulation environment to the MOKA Framework.

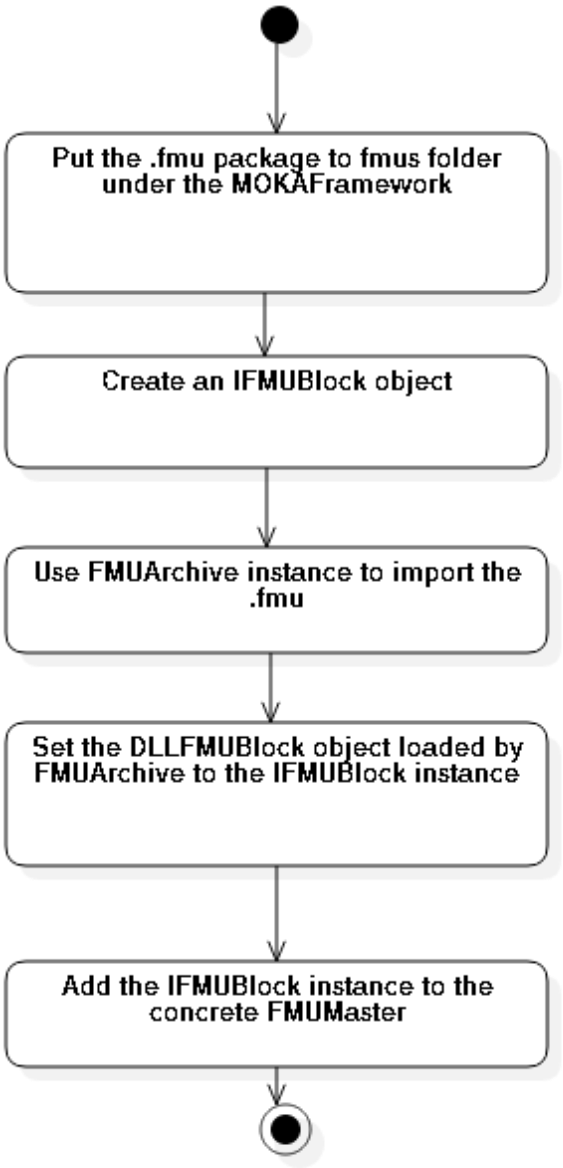

Figure 16 – Activity diagram for loading an FMU to the MOKA Framework

5.3 Constructing an FMUMaster

In this sub-section development of an FMU Master using MOKA Framework is described.

The concrete master that will be used throughout the simulation is to be defined as a class that inherits from the FMUMaster. As described in section 2.3.4 Computational Flow of FMU Co-simulation and section 4.3.9 FMUMaster, basic structure of a co-simulation master is initializing the slaves, triggering the computation and handling proper termination of the simulation. Thus, for the concrete master to specialize these steps of co-simulation; it needs to override the virtual functions for initialization, computation and termination steps.

5.4 Co-Simulation of FMUs

MOKA Framework provides an infrastructure to perform complex co-simulation of FMUs. The FMUs in the constructed co-simulation environment can be either developed using the MOKA Framework itself or they can be exported from another modeling and simulation tool that support FMI standard.

The basic steps of co-simulation of FMUs using MOKA Framework can be listed as the followings:

- FMUs that are developed in another modeling tool and provided as *.fmu* package from should be imported and given to the concrete FMUMaster.
- FMUMaster should be triggered to initialize the simulation. In this step, FMUMaster uses FMUs that are developed in MOKA Framework to create model instances.
- The proper integration of all FMUs done at the initialization step of FMUMaster and after the initialization step, FMUMaster should be triggered to begin with calculation steps.
- After the simulation is terminated by the FMUMaster, simulation results can be analyzed.

The listed steps for the co-simulation of FMUs using MOKA Framework are shown in Figure 17 – Sequence diagram for co-simulation of FMUs using MOKA Framework.

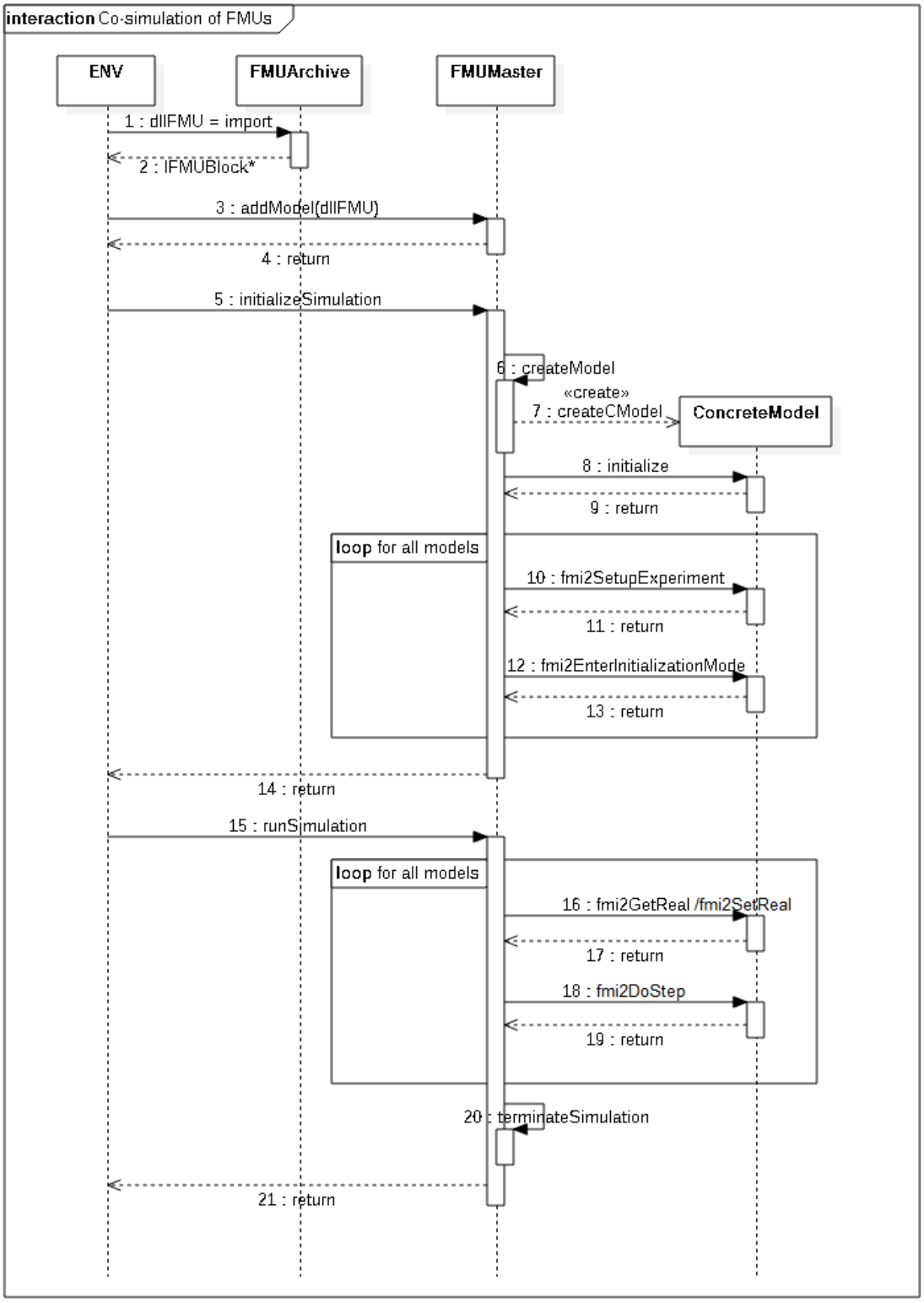

Figure 17 – Sequence diagram for co-simulation of FMUs using MOKA Framework

CHAPTER 6

CASE STUDY

This chapter includes a case study for the demonstration of MOKA Framework in terms of several aspects. In order to test the functionality and usage of the framework, a co-simulation environment which presents an engagement scenario that consists of a flying target and a guided missile with a seeker is developed. Co-simulation results are evaluated and compared with the results gathered from a trusted simulation tool in order to check the calculation correctness of the MOKA Framework. The case study co-simulation scenario enables us to assess the potential of MOKA Framework as an infrastructure to develop, integrate and co-simulation of FMUs.

The framework is further analyzed in terms of FMU development process and framework overhead. In order to demonstrate FMU development process, the process steps are compared with the standard model development steps in order to develop models that conform to the FMI specification. With this comparison it can be tested that MOKA Framework has some potential to promote code reuse and hiding the details of FMU development process while guiding the developer. Additionally, the execution time overhead due to framework usage is assessed. This is meant to illustrate the estimation process for the user's own execution environment.

6.1 Simulation Overview

In order to demonstrate the usage of MOKA Framework and observe the functionality of the framework, a co-simulation environment that represents an engagement scenario of a target and missile system is developed. This co-simulation environment consists of a Kinematic Target model that represents a flying target, a Kinematic Missile target that represents a guided missile with a seeker, a Guidance model that is the guidance block of the missile, an Autopilot model that represents

the autopilot of the missile, the seeker block of the missile and the fuse model for the target hit.

The simulation is designed to evaluate the functionality of four main usage of the MOKA Framework, namely, (i) construction of new FMUs using the framework, (ii) loading FMUs from other modeling tools, (iii) constructing an FMUMaster and (iv) co-simulation of integrated models.

6.2 Simulation Models

This section describes the simulation models. Each model is described with its roles and responsibilities during the simulation, input and output connections, and its interactions with other models in the environment. The simulation environment models and their interactions with each others can be observed in Figure 18 – Case study model dependencies.

6.2.1 Kinematic Missile Flight Model

Kinematic Missile Flight Model represents the behavior of the missile as time passes. Initial position and velocity of the missile are given to this model as the input in the initialization phase. Fin deflections which are obtained from Autopilot model are the other input of this model which is used to calculate lateral aerodynamic force. Kinematic Missile Flight Model updates the position and velocity of the missile. Position of the missile is transmitted to Fuse Model. Autopilot model also calculates the Euler angles and Direction Cosine Matrix of the body relative to the earth. While Euler angles are connected to the Guidance and Autopilot model, Direction Cosine Matrix is just fed to the Autopilot model. The last output of Kinematic Missile Flight Model is the acceleration of the missile which is connected to the Autopilot model.

6.2.2 Kinematic Target Flight Model

Kinematic Missile Flight model shows the characteristic behavior of the missile with the time. Initial position and velocity of the target are given to this model as the input in the initialization phase. Instant position and velocity of the target are obtained as

the outputs. While position of the target is sent to Fuse and Seeker model, velocity of the target is just used in the Seeker model.

6.2.3 Guidance Model

This block calculates the acceleration command in order to maneuver the missile to intercept the target. To achieve this maneuver proportional guidance law algorithm is implemented. This algorithm calculates the commanded acceleration proportionally to the line of sight change between the missile and target. This block takes the position of missile and target from autopilot and seeker block and calculates the line of sight change of the target. Then it compensates the gravitational acceleration.

6.2.4 Autopilot Model

This block estimates the position and velocity of the missile which is used in the Seeker model. By using the Direction Cosine Matrix and Euler angles from Kinematic Missile Flight model, Autopilot model calculates proper total control input for the two axes of the missile which are rudder and elevator for the given acceleration command from guidance block. Rudder command is used to maneuver in yaw direction and elevator command is for pitch maneuver. Afterwards, these total maneuver command are distributed to four different control surfaces which is transmitted to the Kinematic Missile Flight model.

6.2.5 Seeker Model

This model computes line of sight rate and velocity of the target relative to the missile. For this purpose, velocity and position of both missile and target from Autopilot and Kinematic Target model respectively, are given as the inputs of this model. The outputs of the Seeker model are connected to the Guidance model.

6.2.6 Fuse Model

Fuse model is responsible for activating warhead mechanism when the distance between missile and target gets smaller than predetermined value. Fuse model takes missile position from Kinematic Missile Flight model and target position from Kinematic Target Flight model as inputs. It works as binary state mechanism and

gives fuse situation as the output. If the distance between missile and target gets smaller than predetermined value, the binary output gets on and the simulation stops. Otherwise, the simulation keeps on running.

6.3 Importing FMU Models

In order to demonstrate the availability of heterogeneous co-simulation environment of MOKA Framework, some of the afore-mentioned models are developed in another modeling tool and some of them are developed using MOKA Framework itself.

Following FMU slaves are the developed in MATLAB/Simulink® [16] exported as FMU structure by using the FMI Toolbox for MATLAB/Simulink® from Modelon [5] :

- Kinematic Missile Flight Model
- Kinematic Target Flight Model
- Guidance Model
- Autopilot Model

On the other hand, Fuse and Seeker models are developed using the MOKA Framework. Details of this process will be given in the following section.

In order to import FMUs taken from the toolbox, steps listed in section 5.2 Loading an FMU of the chapter 5 MOKA FRAMEWORK USAGE are followed. For each model, an IFMUBlock instance is created. FMUArchive instance is triggered for the importing process. FMUArchive extracts the *.fmu* package, loads the shared library of the FMU, creates an instance of the DLL_FMUBlock and transmits the handle of the loaded shared library to this instance. The IFMUBlock instances are set to each DLL_FMUBlock accordingly. The imported instances are then added to the FMU list of the simulation master.

6.4 Constructing Fuse and Seeker Models

Fuse and Seeker models are developed using the MOKA Framework itself and the slaves are added to the simulation environment to co-simulate with other FMUs. On the development process, steps listed in the section 5 of the chapter 5 MOKA FRAMEWORK USAGE are followed. An FMUPort to hold missile position, target position and fuse situation is created. Since the fuse situation reflects the state of the Fuse model, it is defined as a StateVariable.

For the Fuse model, a concrete class that inherits from FMUBlock class is created. *setStartValues* and *stepAction* virtual functions are overridden in order to reflect the Fuse model behavior. After the concrete instance is initialized, the FMUPort instance and the FMUStateVariables instance that holds the fuse situation are connected to the Fuse model block.

Same steps are followed for the Seeker model. An FMUPort to hold missile position, missile velocity, target position, target velocity, Line of Sight (LOS) rate and velocity of the missile relative to the target is created. Since LOS rate and velocity of the missile reflect the state of the Seeker model, they are defined as StateVariable. To represent the model block, a concrete class that inherits from FMUBlock class is created. *setStartValues* and *stepAction* virtual functions are overridden in order to reflect the Seeker model behavior. After the concrete instance is initialized, the FMUPort instance and the FMUStateVariables instance that are created for the Seeker model are connected to the Fuse model block.

Codes for the development process of Fuse and Seeker models are given in Appendix B, CASE STUDY CODE SNIPPETS

6.5 Integration of Models

After the models that are listed in section 6.3 Importing FMU Models is imported and Fuse and Seeker models are developed, the next step is to prepare these FMU slaves for the co-simulation. The first step of this process is to integrate the slaves by considering input and output dependencies.

Figure 18 shows the input and output dependencies of the FMU slaves in the case study simulation environment.

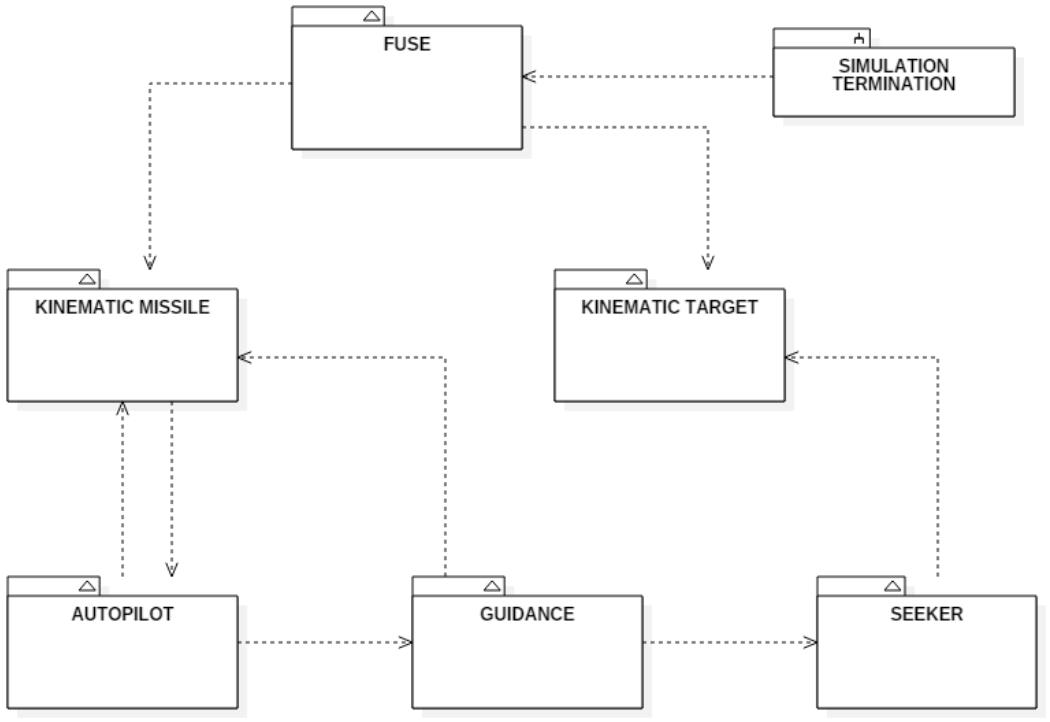

Figure 18 – Case study model dependencies

Considering the data dependency between FMU slaves, proper connection of inputs and outputs completes the part of model integration.

At the beginning of each computation step, concrete FMUMaster gathers the outputs from each of the slaves, and sets the inputs for the preparation of next computation step. After the integration of the slaves with such a configuration of input and output dependencies, each slave is triggered to perform computation with their own solver.

The computation order is important for this co-simulation scenario. The first model to perform calculation with its own solver is the Kinematic Target Flight model. It computes the velocity and position of the target. After the target model, the second model to perform calculations is the Kinematic Missile Flight model. The model computes the position and velocity of the missile.

Additionally, it calculates the Euler angles and Direction cosine matrix of the body relative to the earth. After these two models performed calculations, Fuse, Seeker, Guidance and Autopilot components that guide the flight of the missile perform calculations.

At the end of one computation step, concrete FMUMaster check the fuse situation state variable of the Fuse model in order to detect termination of the simulation. If the termination condition is met, the slaves are triggered to terminate properly, and if the termination condition is not met, the communication i.e. input and output transfers are done and the cycle goes again.

6.6 Evaluation

This section evaluates the MOKA Framework in terms of the results of constructed simulation environment for case study, comparison of FMU development process with MOKA Framework and manual FMU development. Further, run-time performance overhead of the framework is assessed in order to show how to calculate the execution time overhead of using the framework in a particular execution environmnet.

6.6.1 Case Study Scenario Results

The constructed simulation environment demonstrates the usage of MOKA Framework in terms of several aspects: importing models that are developed in another modeling tool, developing FMUs with the framework itself and performing integration and co-simulation of these heterogeneous FMU slaves. Besides these demonstrations, the calculation correctness of the MOKA Framework is needed to be evaluated. To observe the calculation correctness of the MOKA Framework, the case study co-simulation scenario is performed as described in the previous sections of these chapter and the results of the simulation is compared to the results gathered from MATLAB/Simulink® [16] simulation tool.

The co-simulation represents an engagement scenario with a helicopter and a guided missile. The helicopter starts movement at 2000 meters altitude and flies with a

speed of 50 m/h while the missile starts movement from the ground. The missile reaches the maximum speed of 620 m/h during the simulation. The missile approaches to the helicopter at each time step. When the distance between the missile and the helicopter gets smaller than 5 meters, the missile activates the warhead mechanism and the simulation terminates. The simulation runs for 18598 computations steps with the step size of one millisecond. The elapsed time (wall-clock time) for this simulation run is around 21seconds.

The developed fuse and seeker models are instantiated and configured with these initial parameters. The engagement FMU master then imported the Kinematic Missile, Kinematic Target, Guidance and Autopilot models. After the import phase, the initial parameters are set for these models by the engagement master. The communication point duration is 1 millisecond. During two communication points all of the FMU slaves compute their part for the realization of engagement scenario. After each computation time step, engagement master checks the termination condition, which is whether the distance between missile and target is less than 5 meters. For these scenario parameters, the missile hits the target after 18598 computation steps. Missile and target positions for each computation time step are logged.

After the MOKA Framework run the co-simulation scenario, the same models and co-simulation scenario settings are run in MATLAB/Simulink® [16]. Missile and target positions for each computation time step is logged for the MATLAB/Simulink® [16] also in order to compare them with MOKA Framework results.

Figure 19 shows the co-simulation results in terms of target and missile path for both MOKA Framework and MATLAB/Simulink® [16].

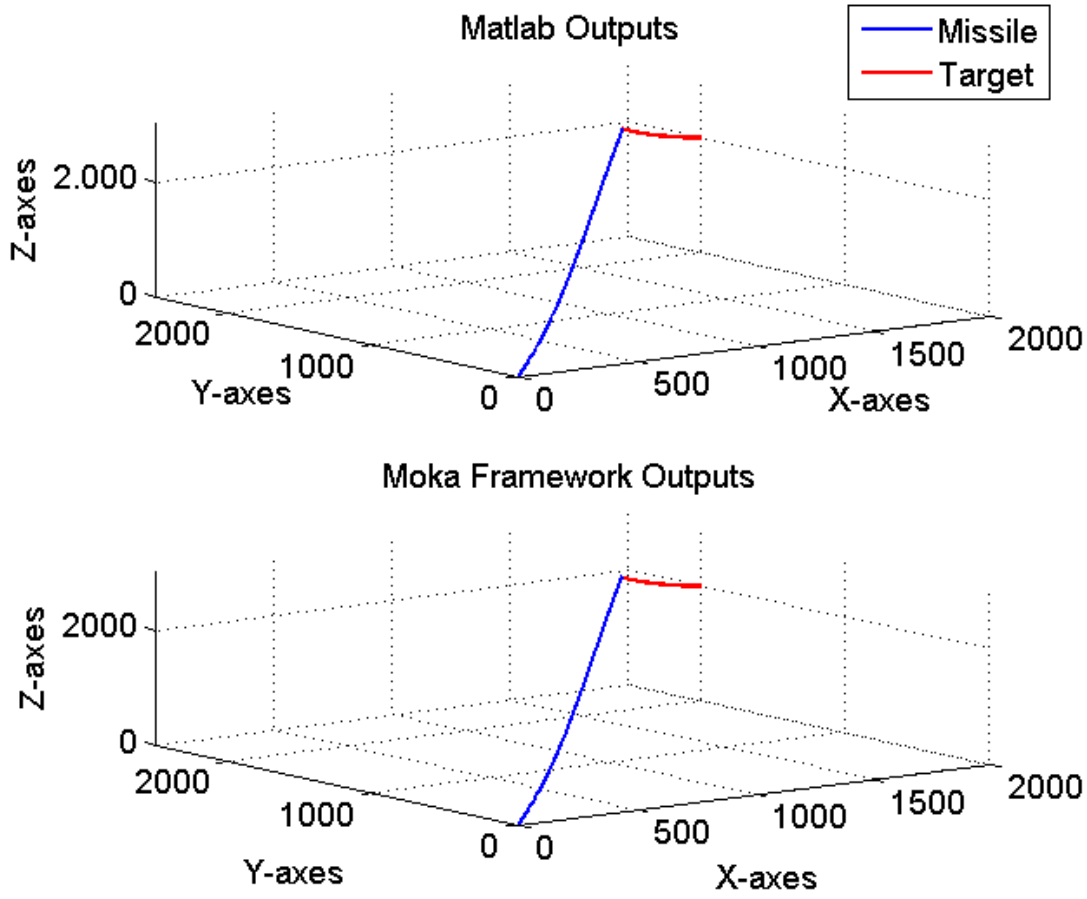

Figure 19 – Engagement results

The missile reaches the target and activates the warhead mechanism at the expected coordinates. There exist some numerical differences between MOKA Framework and MATLAB/Simulink® [16] that are caused by implementations of mathematical definitions such as sinusoidal calculations, double-precision floating-point number representation, etc.

At the end, the results showed that MOKA Framework achieved the expected integration and co-simulation results since a co-simulation scenario with various dynamic system models could be constructed and performed.

6.6.2 FMU Development Process Comparison

In order to evaluate the contributions of MOKA Framework in terms of development of a dynamic system model that conforms to FMI specification, steps of development process with MOKA Framework and traditional FMU development are compared.

Actions needed to be followed in order to obtain a model with FMI compatibility are listed in Table 3. The activities for manual FMU development are gathered from the specification [13] and QTronic FMU SDK [7].

Table 3 – FMU development process comparison - MOKA Framework vs. manual FMU development

FMU Development Process	**Activities with MOKA Framework**	**Activities with Manual Development**
Creation of model	Creation of a class that inherits from FMUBlock Class definition should include *setStartValues* and *stepAction*.	Creation of a new model structure Class definition should include all FMI interface.
FMI interface implementation	Only *setStartValues* and *stepAction* should be implemented.	All FMI interface should be implemented.
Input/Output definition of the model	An instance of FMUPort should be created and variables should be added to this instance.	Array sizes for all variable types should be defined.
		All value references (definitions for array indexes) should be defined.

		The XML file (*modelDescription.xml*) that defines model variables should be created.
State Definition of Model	An instance of FMUStateVariables should be created and filled with StateVariable instances.	Number of state variables and event indicators should be defined.
		All value references for state variables should be defined.

From the Table 3, it can be observed that the number of required activities to develop an FMU with the MOKA Framework is less than manual construction of an FMU. We need to analyze these steps in detail in order to have an insight about the intricacies of the development processes for MOKA Framework and manual FMU development.

In the model creation phase, a class that inherits from the FMUBlock class inside the MOKA Framework is to be defined. This class definition should include *stepAction* and *setStartValues* methods. Similarly, a structure to represent model framework should be defined. This model structure should include all methods in the FMI interface, since there is no inheritance for the definitions of these methods as in the MOKA Framework. If available, existing code could be in reused copy-and-paste fashion –an error prone practice. Hence, it can be claimed that the definition of a model structure is a bit easier and robust in MOKA Framework thanks to inheritance.

For the realization of defined model structures, proper FMI interface implementations must be provided. In the MOKA Framework, *stepAction* and *setStartValues* methods are to be implemented. The *stepAction* method includes

actions that represent the calculation algorithm of the model while the *setStartValues* method includes actions for the initial parameter settings of the model.

The FMI interface methods are implemented in the FMUBlock base class; hence the model reuses them and does not need to implement the specification again. On the other hand, in the manual development, all FMI methods are needed to be implemented. These implementations also cover the implementation of *stepAction* and *setStartValues* methods because the actions for the computation part and the actions for the initial parameter settings remain as to be implemented again. Hence, it can be claimed that the realization of model structures step requires less effort in the MOKA Framework compared to manual FMU development due to code reuse from base class FMUBlock.

For the input/output definition of the model, in the MOKA Framework an instance of the FMUPort class should be created and each variable should be added to this instance. These variables are added to the collections with the proper data types and the value references for these variables are created automatically inside the FMUPort. On the other hand, the collections for the input/output variables of the model should be created and initialized manually in the manual FMU development case. Since all inputs and outputs with the same data type are kept in a single array in the manual FMU development form, the corresponding indexes or value references for the variables are needed to be defined. This action is relatively time consuming and it is error-prone since the indexes of the variables can be easily mixed up, which result in with reaching a wrong value for a particular variable. Furthermore, in the manual FMU development case, an XML file that describes the input/output variables and value references needs to be created for the model.

For the state variables definitions of the model, in the MOKA Framework an instance of the FMUStateVariables class should be created and each state variable should be added to this instance. The number of state variables is kept inside the FMUStateVariables instance. For the manual development of FMUs, the collections for the state variables of the model should be created and initialized manually. Again,

the corresponding indexes or value references for the state variables are needed to be defined that is a time-consuming and error-prone activity as described. Additionally, number of state variables and event indicators should be defined in the manual FMU development case.

Based on these comparisons of FMU development processes for MOKA Framework and manual FMU development, it can be argued that MOKA Framework offers a potential to promote code reuse with the help of inheritance and ease the development process by hiding implementation details of the FMU development process from the user.

We undertook a relatively simple case study to have an empirical result on the FMU development process comparison. In order to compare the required development time considering the listed steps in Table 3, a software engineer conducted the FMU development process for the Fuse Model, which involves a relatively simple algorithm. The software engineer, Faruk Yılmaz, is familiar with the FMI interface and he is the author of *Adapting Functional Mockup Units for HLA-compliant Distributed Simulation* paper [27]. Firstly, he constructed the Fuse model according to the steps to develop an FMU using the MOKA Framework. The development process took approximately 5 hours to complete. The time that is needed to learn the Fuse model algorithm and characteristics is included to this development time. Secondly, he developed the Fuse model by implementing FMI methods for the Fuse model algorithm after a day off. The development process took approximately 1.5 days (8 hours of work per day). The difference is caused by the fact that he had to start from scratch for the development of an FMU in the manual development case while he had a template FMU in the case of FMU development with MOKA Framework. This assessment gives a preliminary indication about the potential of MOKA Framework in required-time improvement on FMU development process. This experiment was a small-scale experiment to assess the development process performance. It should be replicated with more developers and case studies. More

importantly, framework usage metrics should be systematically gathered in the course of MOKA framework regular use [35].

6.6.3 Framework Overhead

MOKA Framework introduces an object-oriented approach to the development and co-simulation of FMU slaves. The framework differentiates these processes from the ones performed by pursuing original FMI specification in terms of several topics.

Firstly, the development and simulation language changed from C programming language to C++. The C++ language is chosen in order to achieve the object-oriented design, and whether the language change causes the framework to incur an overhead is beyond the scope of the present evaluation. The issue has already been investigated with the analysis of two languages in terms of performance for different configurations [25][26][28].

Secondly, since MOKA Framework proposes an object-oriented design, the performance impact can be on object construction, object destruction, inheritance and dynamic method invocation. The main comparison is of these topics are again beyond the scope of the framework overhead since the issue is now object-oriented programming versus procedural (structured) programming.

One paradigm that can be investigated to approximate the overhead of the MOKA Framework is to evaluate the use of dynamic method invocation, i.e. virtual function calls. The metric of this evaluation will be the time overhead of using of a single virtual function call compared to the zero virtual function call for exactly the same calculation. This way it can be measured that if the MOKA Framework was not provide virtual function in order to have templates such as FMU slave and state variables, but provides concrete classes for them any overheads would be avoided.

Even if the C to C++ language overhead, object-orientation overhead and virtual function overhead exceeds the scope of a specific framework overhead but rather can be investigated in terms of structural software development approaches versus object-oriented methodologies, total effect of these paradigms on MOKA

Framework can be observed with an isolated simulation run for a model. We propose an assessment for the execution time of a dynamic system model in order to show how to calculate the execution time overhead of using the framework in a particular execution environment.

For this purpose, an application to record execution time differences of a model developed by the MOKA Framework and a model developed manually is constructed. The Fuse model that is described in section 6.2.6 is developed with MOKA Framework and manual FMU development steps. Both models run with the same simulation settings for 18598 computations steps with the step size of one millisecond. The execution times are recorded for each model. The execution time measurement is repeated 10 times and these results are given in Table 4.

Table 4 – Model execution time overhead test results for a single step

Run Number	**Fuse Model Execution Time (MOKA Framework) (microseconds)**	**Fuse Model Execution Time (Manual FMU Development) (microseconds)**	**Execution Time Difference (microseconds)**
1	0,165853	0,141600	0,0242529
2	0,213005	0,184477	0,0285288
3	0,176759	0,153749	0,0230099
4	0,203921	0,184656	0,0192645
5	0,167331	0,137955	0,0293753
6	0,162713	0,133942	0,0287717

7	0,165787	0,133531	0,0322557
8	0,166199	0,143118	0,0230803
9	0,202305	0,184710	0,0175948
10	0,169593	0,141926	0,0276671

The machine configuration for the given performance results is listed in Table 5.

Table 5 – Machine configuration for virtual function overhead runs

CPU	Intel Core i7-2630QM CPU 2.00 Ghz 4 Core(s), 8 Logical Processor(s)
RAM	8 GB 256 KB L2 Cache 6144 KB L3 Cache
Virtual Memory	15.9 GB
Page File Space	7.95 GB
OS	Windows 7 Home Premium 64 Bit 6.1.7601 Service Pack 1 Build 7601
Compiler	Microsoft <R> C/C++ Optimizing Compiler Version 17.00.50727.1 for x86

From the runs, it can be seen that the average execution time overhead caused by the framework usage for this specific scenario was 0.025 microseconds for a single

calculation step. The execution time ratio framework FMU over manually developed FMU is 1.16.

The important point for such assessments is the number of samples, i.e., the number of runs. For this purpose, it should be computed how close the sample mean should be compared to true mean. In order to show that the numbers of samples, i.e., runs, are enough and the results are close to true mean, confidence interval for the sample mean is computed using the procedure presented in chapter 4 of [32]. With the calculations, the sample mean is within $\pm 0{,}31\%$ different from the true mean with a confidence interval level of *99%* (with a probability of *99%*). The number of runs can said to be sufficient for this experiment.

The details of confidence interval calculation are given in APPENDIX C: CALCULATION DETAILS OF CONFIDENCE INTERVAL FOR .

This execution time difference is caused by C to C++ language overhead, object-orientation and virtual function calls. The first two are constant for a particular simulation scenario but the third one, virtual function calls, change with the number of calls of virtual functions defined in the MOKA Framework. For example, if the simulation developer begins to call more FMI functions such as status checks for the model, the execution time overhead increases. For this purpose, virtual function call cost of the MOKA Framework is expressed with an assessment. In order to approximate such overhead observation, an application to record a virtual function overhead is developed. For the first part, a Base class is created so that it looped until the end of the simulation with calling the virtual helper function. The virtual function is realized in the concrete class and this concrete class is used in the experiment for the performance measurement of virtual function overhead. Then, for the opposite part, a class that does exactly the same calculation loop is implemented with the elimination of virtual function calls. In this application, the computational task is to calculate the 20th Fibonacci number iteratively for 1000 times. This configuration was to have a compute-bound function in order to observe the performance results more clearly. The performance of the calculation functions for these two cases are

recorded and evaluated. The performance measurement repeated 10 times and the result for each run is given in Table 6. Run configurations are exactly the same.

Table 6 – Virtual function overhead test results for a single step

Run Number	**Calculation Performance with Virtual Function Call (microseconds)**	**Calculation Performance without any Virtual Function Calls (microseconds)**	**Additional Cost of a Virtual Function Call (microseconds)**
1	0,103656	0,101604	0,002052
2	0,087235	0,078512	0,008723
3	0,098525	0,078512	0,020013
4	0,079538	0,078512	0,001026
5	0,079025	0,078512	0,000513
6	0,079025	0,078512	0,000513
7	0,079538	0,077999	0,001539
8	0,079025	0,078512	0,000513
9	0,081591	0,077999	0,003592
10	0,079025	0,078512	0,000513

The machine configuration for the given performance results is listed in Table 5 – Machine configuration for virtual function overhead runs. The computation times was measured via the number of CPU cycles divided by the CPU cycle time. The

execution times were exactly the same for some runs since the computational task is not much complex and CPU cycles were same.

From the runs, it can be seen that a single virtual function call overhead is on the average 0.004 microseconds. The confidence interval of the run results are computed as $\Delta=2.59484$. Hence, the sample mean is $\pm 2{,}59\%$ of the true mean with a confidence level of *99%* (with a probability of *99%*). The number of runs can said to be enough for this experiment. The details of confidence interval calculation are given in APPENDIX C: CALCULATION DETAILS OF CONFIDENCE INTERVAL FOR

Hence, for this specific case study we can say that we measured the additional cost of a virtual function call (ACVFC) is as the following:

Additional Cost of a single Virtual Function Call:

$$ACVFC = 0.004 \text{ microseconds.} \quad (1)$$

Since the average time overhead result covers only one virtual function, we need to compute the total virtual function calls in the MOKA Framework during a simulation. Table 7 shows the virtual functions and their number of calls for the duration of a complete simulation run.

Table 7 – Virtual function calls in MOKA Framework

Owner Class	**Virtual Function**	**Number of Calls in a Simulation Run**
`BaseStateVariable`	`Update`	`N`
	`Initialize`	`1`
`FMUBlock`	`setStartValues`	`1`

	stepAction	N
IFMUBlock	FMI Computation functions (rather than fmi2DoStep)	1 (in general)

In Table 7, N is the number of steps in the simulation. The number of FMI Computation Functions can vary according to simulation scenario. As seen from Table 7 the MOKA Framework virtual function overhead can be calculated as in the following:

Total Number of Virtual Function Calls:

$$VFC = 2*(N+1) + \text{Number of FMI Computation Function Calls} \quad (2)$$

Virtual Function Overhead of MOKA Framework:

$$VFO = VFC * ACVFC \text{ (in microseconds)} \quad (3)$$

This representation gives an insight into the execution time overhead caused by the virtual function calls of the MOKA Framework. The value of additional cost of a single virtual function call changes with the machine configuration, operating system, the compiler, and the co-simulation scenario itself. Hence in order to measure the specific performance time overhead, it should be measured in the simulation environment. Similar experiments should be repeated and additional cost should be determined for the non-virtual function calls for designed specific simulation environment.

To sum up, MOKA Framework causes some execution time overhead when compared with manual simulation environment development with FMI. The main causes for this situation is the C to C++ language overhead, object-oriented structure of the MOKA Framework and the virtual function calls that come from the generic FMU structure of MOKA Framework. Even if the paradigms that causes the execution time overhead should be considered in terms of structural software

development approaches versus object-oriented methodologies, to observe the total effect of these paradigms on MOKA Framework some assessments are performed. These assessments are performed to show how to calculate the execution time overhead of MOKA Framework usage compared to manual simulation environment usage for FMI. The results are specific to machine configuration as well as simulation settings. The developers of simulation environment who want to have an insight about the execution time overhead can follow these steps and make specific assessments for their own simulation settings and machine configurations.

CHAPTER 7

CONCLUSION

This thesis work introduces an object-oriented framework for FMI co-simulation, named MOKA. The FMI model interface standard is being evaluated by the modeling and simulation community due to its promise for seamless integration and co-simulation of models. There is a tendency to develop applications to interact with models that are compatible with FMI specification. However current tools are still in progress and there is still a need for well-established applications to develop, integrate and co-simulate the models that are compatible with the FMI specification for object-oriented development methodologies. At this point, MOKA Framework aimed to bridge the gap between FMI specifications and object-oriented modeling and simulation needs such as modularity, code reuse and extendibility.

The MOKA Framework introduces an infrastructure for the development, integration and co-simulation of FMUs. Firstly, the FMI for Co-simulation version 2.0 is mapped to a C++ API to support object-orientation for the development environment. Secondly, MOKA Framework provided a modular structure for the development, integration and co-simulation of FMUs. The roles of an FMU during simulation are divided into two parts: a component to manage input/output interactions, and a component to perform calculation with the solver of the model. These components are represented with class structures as FMUPort and IFMUBlock accordingly. IFMUBlock represented the interface for both of the models that are developed using the framework itself and that are generated from another modeling tool and will be imported to the MOKA environment for co-simulation. The state variables of such model blocks that are developed using the MOKA Framework are represented with abstract BaseStateVariable and concrete StateVariable classes. Thirdly, for the integration and co-simulation of the FMUs, MOKA Framework introduces an FMUMaster template to facilitate the implementation of the master-slave architecture for co-simulation. This master is extended by the user for the

specification of particular co-simulation settings. FMUArchive, a helper class of the FMUMaster, is used for importing foreign FMUs into the MOKA environment.

MOKA Framework contributes to modeling and co-simulation with FMUs since it provides an infrastructure for the development, integration and co-simulation of the models in a single environment. It supports code reuse by providing templates for FMU slaves and the co-simulation master model. The simulation developers can take the advantage of the guidance in model and master development complying with FMI specification. In addition, MOKA Framework promotes the adoption of FMI specification with the advantages of object-oriented methodologies such as modularity and extendibility. The framework provides a modular structure for the models and the master FMU, so that developers can construct specialized subsystems and obtain various system versions by combining different subsystems.

The capability of the MOKA Framework is demonstrated in practice with an engagement scenario that consists of a flying target and a guided missile with a seeker. This demonstration went through the usage steps of MOKA Framework, namely FMU development, importing FMUs that are generated by other modeling tools and co-simulation by proper integration of the FMU slaves.

For the FMU development demonstration Fuse and Seeker components of the guided missile are developed using the MOKA Framework and the steps are described. For the import process demonstration of foreign FMUs, Kinematic Missile Flight model, Kinematic Target Flight model, Autopilot model and Guidance model are developed in MATLAB/Simulink® [16] and imported to FMU structure by using the FMI Toolbox for MATLAB/Simulink® from Modelon [5]. The usage of the helper class FMUArchive is discussed. Finally, the input/output dependencies of the simulation models are accounted and the integrated models are triggered to compute the co-simulation results. The results are compared with the results obtained from the MATLAB/Simulink® [16] simulation tool in order to check the functional correctness of the MOKA Framework. The case study co-simulation scenario

suggests that the MOKA Framework holds promise to serve as an infrastructure for development, integration and co-simulation of FMUs.

MOKA Framework further evaluated in terms of contributions on FMU development process compared to traditional FMU development without using the framework. For the comparison the steps of processes are listed and discussed. The comparison of suggested that the MOKA Framework has a potential to promote code reuse with the help of designed base model, the FMUBlock, and ease the development process by hiding implementation details of the FMU development process from the user in the definitions of classes.

The framework is further evaluated in terms of its run-time overhead. For the estimation of framework execution time overhead, assessments made that compares the execution time of a model developed with MOKA Framework and same model developed manually for FMI. The execution time overhead of the MOKA Framework is analyzed in terms of virtual function calls. The assessments are performed to show how to calculate the execution time overhead of MOKA Framework usage compared to manual simulation environment usage for FMI for a particular simulation setting and machine configuration.

The prospects for future research may to extend the co-simulation mechanism in such a way that a methodology to construct and use co-simulation algorithms as external finite state machines. There is an ongoing study to run different co-simulation scenarios without specifying a concrete FMU master implementation. The generic FMUMaster instance may interpret the co-simulation scenarios, i.e. finite state machine scripts that are written in a domain specific language and run the co-simulation. This approach has the potential to reduce the need to generate a specific concrete FMUMaster instance for various co-simulation scenarios. In addition to this, MOKA Framework is indented to be extended to support hardware in the loop simulation and to provide functionalities for real-time constraints on a co-simulation scenario. Another future research may be to extend the FMU development process by

integrating a graphical user interface to the framework. This way the framework has the potential to ease the development of FMUs.

REFERENCES

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice* (1st ed.). Morgan & Claypool Publishers.

[2] Christian Bertsch, Elmar Ahle, Ulrich Schulmeister (2014). The Functional Mockup Interface - seen from an industrial perspective, Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden

[3] Chen, W., Huhn, M., and Fritzson, P. A generic FMU interface for Modelica. In Proc. EOOLT (2011).

[4] Henningsson, T. M., A¨ kesson, J., and Tummescheit, H. An FMI-based tool for robust design of dynamical systems. In In Proc. 10th International Modelica Conference (2014).

[5] FMI Toolbox for MATLAB/Simulink®, Modelon (2014). Retrieved January 21, 2016, from http://www.modelon.com/products/fmi-toolbox-for-matlab/

[6] JModelica, FMI Library (FMIL) (2014). Retrieved January 21, 2016, from http://www.jmodelica.org/FMILibrary/

[7] Qtronic Software Development Team, FMU SDK: Free Development Kit (2014). Retrieved January 21, 2016, from http://www.qtronic.de/en/fmusdk.html

[8] Widl, E., Muller, W., Elsheikh, A., Hortenhuber, M., and Palensky, P. The FMI++ library: A high-level utility package for FMI for model exchange. In Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on (May 2013), 1l–6.

[9] J. van Gurp and J. Bosch. 2001. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software: Practice and Experience*. 31, 3 (March 2001), 277-300.

[10] Kirk, D., Roper, M., & Wood, M. (2006). Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering Empir Software Eng*, 12(3), 243-274.

[11] Mohamed Fayad and Douglas C. Schmidt. 1997. Object-oriented application frameworks.*Commun. ACM* 40, 10 (October 1997), 32-38.

[12] Ralph E. Johnson. 1997. Components, frameworks, patterns. In *Proceedings of the 1997 Symposium on Software Reusability* (SSR '97), Medhi Harandi (Ed.). ACM, New York, NY, USA, 10-17.

[13] MODELISAR Consortium (2012).Functional Mock-up Interface for Model Exchange and Co-Simulation. Retrieved January 21, 2016, from https://www.fmi-standard.org/

[14] Neema, H., Gohl, J., Lattmann, Z., Sztipanovits, J., Karsai, G., Neema, S., . . . Sureshkumar, C. (2014). Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden.*

[15] Bastian, J., Clauß, C., Wolf, S., & Schneider, P. (2011). Master for Co-Simulation Using FMI. *Proceedings from the 8th International Modelica Conference, Technical University, Dresden, Germany.*

[16] MATLAB, SIMULINK (2006). Retrieved January 21, 2016, from http://www.mathworks.com/products/simulink/

[17] SciLab, Xcos (n.d.). Retrieved January 21, 2016, from https://www.scilab.org/scilab/gallery/xcos

[18] JFMI - A Java Wrapper for the Functional Mock-up Interface. (2012). Retrieved January 21, 2016, from http://ptolemy.eecs.berkeley.edu/java/jfmi/

[19] Demming, R., & Duffy, D. J. (2010). *Introduction to the Boost C libraries. Amsterdam,* Volume 1, Foundations. The Netherlands: Datasim Education BV.

[20] Shriver, B. D., & Wegner, P. (1987). *Research Directions in Object-oriented Programming.* Cambridge, MA: MIT Press.

[21] Stroustrup, B. (1997). *The C++ programming Language.* MA: Addison-Wesley.

[22] 7-Zip. (2015). Retrieved January 21, 2016, from http://www.7-zip.org/.

[23] TinyXML-2. (2014). Retrieved January 21, 2016, from http://www.grinninglizard.com/tinyxml2/

[24] Memduha Aslan, Halit Oğuztüzün, Umut Durak, and Koray Taylan. 2015. MOKA: an object-oriented framework for FMI co-simulation. In *Proceedings of the Conference on Summer Computer Simulation* (SummerSim '15), Saurabh Mittal, Il-Chul Moon, and Eugene Syriani (Eds.). Society for Computer Simulation International, San Diego, CA, USA, 1-8. ISBN: 978-1-5108-1059-4.

[25] Viega, J., J.t. Bloch, Y. Kohno, and G. McGraw. "ITS4: A Static Vulnerability Scanner for C and C++ Code." *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00).*

[26] Detlefs, D., Dosser, A., & Zorn, B. (1994). Memory allocation costs in large C++ and C programs. *Softw: Pract. Exper. Software: Practice and Experience,* 24(6), 527-542.

[27] Faruk Yılmaz, Umut Durak, Koray Taylan, and Halit Oğuztüzün. "Adapting Functional Mockup Units for HLA-compliant Distributed Simulation." *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden (2014).*

[28] Rotithor, H., Harris, K., & Davis, M. (1999). Measurement and Analysis of C and C++ Performance. Digital Technical Journal, 10(1), 32-47.

[29] Rabenstein, Rudolf, and Stefan Petrausch. “Block-Based Physical Modeling with Applications in Musical Acoustics”. *International Journal of Applied Mathematics and Computer Science*. Volume 18, Issue 3, Pages 295–305, ISSN (Print) 1641-876X, DOI: 10.2478/v10006-008-0027-6, October 2008.

[30] Thramboulidis, K, Perdikis, D, Kantas, S. “Model driven development of distributed control applications”. *The International Journal of Advanced Manufacturing Technology*. Volume 33, Issue 3, Pages 233-242, ISSN (Print) 1433-3015, DOI: 10.1007/s00170-006-0455-0, 2006.

[31] Latih, R., Patel, A., & Zin, A. M. “The design of Block-Based mashup tool for end users mashup applications development”. *Journal of Theoretical and Applied Information Technology*, Volume 34, Issue 2, Pages 215-225, 2011.

[32] Ropella, Kristina M. Introduction to Statistics for Biomedical Engineers. San Rafael, CA: Morgan & Claypool, Pages 33-37, 2007.

[33] Bernard P. Zeigler. 1984. Theory of Modelling and Simulation. Krieger Publishing Co., Inc., Melbourne, FL, USA.

[34] Hause, Matthew. "OMG Systems Modeling Language (OMG SysML™) Tutorial.", INCOSE International Symposium, 19.1 (2009): 1840-972.

[35] Silva, R. P. E, and E. C. Freiberger. “Metrics to Evaluate the Use of Object Oriented Frameworks”. *The Computer Journal*, Volume 52, Issue 3, Pages 288-304, 2009.

APPENDIX A

FMI CO-SIMULATION INTERFACE FUNCTIONS

The FMI 2.0 co-simulation functions definitions can be observed in this appendix. IFMUBlock class includes these functions definitions. They are realized in FMUBlock class and loaded from the shared library of imported FMU in the DLL_FMUBlock class.

```
// FMU 2.0 Co-Simulation functions
virtual fmi2Component fmi2Instantiate(
                    fmi2String instanceName,
                    fmi2Type fmuType,
                    fmi2String fmuGUID,
                    fmi2String fmuResourceLocation,
                    const fmi2CallbackFunctions* functions,
                    fmi2Boolean visible,
                    fmi2Boolean loggingOn) = 0;

virtual void fmi2FreeInstance(fmi2Component c) = 0;

virtual fmi2Status fmi2SetDebugLogging(fmi2Component c,
                      fmi2Boolean loggingOn,
                      size_t nCategories,
                      const fmi2String categories[])= 0;

virtual fmi2Status fmi2SetupExperiment(fmi2Component c,
                      fmi2Boolean toleranceDefined,
                      fmi2Real tolerance,
                      fmi2Real startTime,
                      fmi2Boolean stopTimeDefined,
                      fmi2Real stopTime) = 0;

virtual fmi2Status fmi2EnterInitializationMode(
                         fmi2Component c) = 0;

virtual fmi2Status fmi2ExitInitializationMode(
                         fmi2Component c) = 0;

virtual fmi2Status fmi2Terminate(fmi2Component c) = 0;

virtual fmi2Status fmi2Reset(fmi2Component c) = 0;

virtual fmi2Status fmi2GetReal(fmi2Component c,
```

```
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              fmi2Real value[]) = 0;


virtual fmi2Status fmi2GetInteger(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              fmi2Integer value[])= 0;

virtual fmi2Status fmi2GetBoolean(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              fmi2Boolean value[]) = 0;

virtual fmi2Status fmi2GetString(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              fmi2String value[]) = 0;

virtual fmi2Status fmi2SetReal(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              const fmi2Real value[]) = 0;

virtual fmi2Status fmi2SetInteger(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              const fmi2Integer value[]) = 0;

virtual fmi2Status fmi2SetBooelan(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              const fmi2Boolean value[]) = 0;

virtual fmi2Status fmi2SetString(fmi2Component c,
                              const fmi2ValueReference vr[],
                              size_t nvr,
                              const fmi2String value[]) = 0;

virtual fmi2Status fmi2GetFMUStateVariable(
               fmi2Component c,
               fmi2FMUStateVariable* FMUStateVariable
                 ) = 0;

virtual fmi2Status fmi2SetFMUStateVariable(
               fmi2Component c,
               fmi2FMUStateVariable* FMUStateVariable) = 0;
```

```
virtual fmi2Status fmi2FreeFMUStateVariable(
            fmi2Component c,
            fmi2FMUStateVariable* FMUStateVariable) = 0;


virtual fmi2Status fmi2SerializedFMUStateVariableSize(
            fmi2Component c,
            fmi2FMUStateVariable FMUStateVariable,
            size_t *size) = 0;


virtual fmi2Status fmi2SerializeFMUStateVariable(
            fmi2Component c,
            fmi2FMUStateVariable FMUStateVariable,
            fmi2Byte serializedStateVariable[],
            size_t size) = 0;


virtual fmi2Status fmi2DeSerializeFMUStateVariable(
            fmi2Component c,
            const fmi2Byte serializedStateVariable[],
            size_t size,
            fmi2FMUStateVariable* FMUStateVariable) = 0;


virtual fmi2Status fmi2GetDirectionalDerivative(
            fmi2Component c,
            const fmi2ValueReference vUnknown_ref[],
            size_t nUnknown,
            const fmi2ValueReference vKnown_ref[],
            size_t nKnown,
            const fmi2Real dvKnown[],
            fmi2Real dvUnknown[]) = 0;


virtual fmi2Status fmi2SetRealInputDerivatives(
                  fmi2Component c,
                  const fmi2ValueReference vr[],
                  size_t nvr,
                  const fmi2Integer order[],
                  const fmi2Real value[]) = 0;

virtual fmi2Status fmi2GetRealOutputDerivatives(
                   fmi2Component c,
                   const fmi2ValueReference vr[],
                   size_t nvr,
                   const fmi2Integer order[],
                   fmi2Real value[]) = 0;
```

```
virtual fmi2Status fmi2DoStep(
          fmi2Component c,
          fmi2Real currentCommunicationPoint,
          fmi2Real communicationStepSize,
fmi2Boolean noSetFMUStateVariablePriorToCurrentPoint) = 0;

virtual fmi2Status fmi2CancelStep(fmi2Component c) = 0;

virtual fmi2Status fmi2GetStatus(fmi2Component c,
             const fmi2StatusKind s,
             fmi2Status* value) = 0;

virtual fmi2Status fmi2GetRealStatus(fmi2Component c,
             const fmi2StatusKind s,
             fmi2Real* value) = 0;

virtual fmi2Status fmi2GetIntegerStatus(fmi2Component c,
             const fmi2StatusKind s,
             fmi2Integer* value) = 0;

virtual fmi2Status fmi2GetBooleanStatus(fmi2Component c,
             const fmi2StatusKind s,
             fmi2Boolean* value) = 0;

virtual fmi2Status fmi2GetStringStatus(fmi2Component c,
             const fmi2StatusKind s,
             fmi2String* value) = 0;
```

Listing 7 – FMU 2.0 co-simulation function definitions

APPENDIX B

CASE STUDY CODE SNIPPETS

For the co-simulation environment in the demonstration of MOKA Framework two models, namely Fuse and Seeker models, are constructed using MOKA Framework FMU development capability. In this appendix, code snippets from the development of these FMUs can be observed.

CONSTRUCTING FUSE MODEL

```
class Fuse : public FMUBlock
{

protected:
      // constant parameter
      double miss_distance;

public:
      // constructor
      Fuse(fmi2String p_instanceName,
            fmi2String p_guid,
            double p_missDistance);

      // destructor
      ~Fuse(void);

      // called by fmi2Instantiate
      virtual void setStartValues();

      // called by fmi2DoStep
      virtual void stepAction();
};
```

Listing 8 – Fuse model code snippet

CONSTRUCTING SEEKER MODEL

```
class Seeker :  public FMUBlock
{
public:
      // constructor
      Seeker(fmi2String p_instanceName, fmi2String p_guid);

      // destructor
      ~Seeker(void);

      // called by fmi2Instantiate
      virtual void setStartValues();

      // called by fmi2DoStep
      virtual void stepAction();
};
```

Listing 9 – Seeker model code snippet

APPENDIX C

CALCULATION DETAILS OF CONFIDENCE INTERVAL FOR SAMPLE MEAN

This appendix shows the calculation details for the confidence interval of sample means that are calculated in section 6.6.3 Framework Overhead. In order to show that the numbers of samples, i.e., runs are enough and the results are close to true mean, confidence interval for the sample mean is computed according to the section 4.3 of the book by Kristina M. Ropella [32] with the following steps:

- The sample mean is computed as the following

  $y_n = \frac{\sum_{i=1}^{n} d_i}{n}$ where y_n is the sample mean, d_i is the time overhead percentage for the run and n is the total number of runs.

- The standard deviation is computed as the following

  $s_n = \sqrt{\frac{\sum_{i=1}^{n} (d_i - y_n)^2}{n}}$ where s_n is the standard deviation.

- How close/realistic the sample mean y_n is with respect to the true mean μ when$n \to \infty$ is computed since we never know the true mean μ as we can never collect infinite number of samples.

  $\Delta = \frac{2 s_n t_g}{y_n \sqrt{n}}$ where Δ is how far the sample mean from true mean, $t_g = 2.58$ is to say g=*99%* of confidence interval.

As a result, our sample average (y_n), is within $\pm\Delta$% of the true mean μ with a confidence level of *99%* (with a probability of *99%*). The number of runs are can be said to be enough for an experiment if the Δ is less than 5.