REAL-TIME OBJECT-ORIENTED FRAMEWORK FOR FMI CO-SIMULATION

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

BY

MERVE ÇAM SİLİK

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

JANUARY 2020

Approval of the thesis:

REAL-TIME OBJECT-ORIENTED FRAMEWORK FOR FMI CO-SIMULATION

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

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Mehmet Halit S. Oğuztüzün
Head of Department, **Computer Engineering**

Prof. Dr. Mehmet Halit S. Oğuztüzün
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering Department, THK University

Prof. Dr. Mehmet Halit S. Oğuztüzün
Computer Engineering Department, METU

Assist. Prof. Dr. Ertan Onur
Computer Engineering Department, METU

Date:10.01.2020

ABSTRACT

REAL-TIME OBJECT-ORIENTED FRAMEWORK FOR FMI CO-SIMULATION

Çam Silik, Merve
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Mehmet Halit S. Oğuztüzün

January 2020, 52 pages

Development of models, their integration into FMI-compliant co-simulation and performing the simulation in real-time environment are crucial tasks in embedded system development.

To introduce an object-oriented co-simulation environment to real-time domain, promote model reuse and minimize effort for co-simulation environment generation a Real-Time Object-Oriented Framework for FMI Co-Simulation was developed for completion of this thesis. A case study which comprises of a control actuation system and sine wave generator is assembled in order to provide a basic example and clarity on how to use the framework. Case study example is compared to classic approach to co-simulation which requires tightly coupling systems and compiling code at every change in data dependency.

The framework is further analyzed in terms of FMU integration process and framework overhead. Execution time overhead is meant to guide user in overhead estimation process.

Keywords: Functional Mock-up Interface, Real-time Co-simulation, Object-Oriented Framework

ÖZ

İŞLEVSEL MAKET ARAYÜZÜ İÇİN GERÇEK ZAMANLI NESNE TABANLI ARAYÜZ ÇATISI

Çam Silik, Merve
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Mehmet Halit S. Oğuztüzün

Ocak 2020 , 52 sayfa

Gömülü sistem geliştirme çalışmalarında modellerin geliştirilmesi, İşlevsel Makt Arayüzü(İMA) uyumlu eş-benzetim ortamlarına entegrasyonu ve gerçek zamanlı ortamda benzetimi kritik seviyede önem taşımaktadır.

Bu tez kapsamında nesne tabanlı uygulama geliştirme yapısının ve İMA uyumlu eş-benzetim ortamlarının gerçek zamanlı sistemlerle entegre hale getirilmesiyle modellerin tekrar kullanımının artması ve gerçek zamanlı test ortamının geliştirme eforunun azalması planlanmaktadır. Çalışma kapsamında geliştirilen çatıya Gerçek Zamanlı Arayüz Çatısı (GAÇ) adı verilmiştir. Çatının kullnımına kontrol tahrik sistemi ve sinüs dalgası simülatöründen oluşan örnek olay senaryosu verilmiştir. Klasik eş-benzetim sisteminden farklı olarak çatı veri bağındaki değişimlerde derleme gerektirmemektedir.

Çalışma kapsamında çatının geliştirme ortamına nesne tabanlı yapısı nedeniyle getirdiği ek yükler hesaplanmış ve kullanıcıya bildirilmiştir.

Anahtar Kelimeler: İşlevsel Maket Arayüzü, Gerçek Zamanlı Eş-Benzetim, Nesne Tabanlı Çatı

To my family

ACKNOWLEDGMENTS

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

I also wish to thank a lot to Prof. Dr. Ece Güran Schmidt for all the valuable knowledge, technical support, academic assistance, innovative ideas.

I also wish to thank a lot to my managers Koray Taylan and Aylin Hatip İpek for all the valuable ideas, efforts to ease the procedural processes and moral support.

I also wish to thank my husband Yusuf Silik for the invaluable support, understanding and encouragements through the process.

I also wish to thank my beloved cat Ceviz for the caring companionship through my sleepless nights.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SYMBOLS

GB	Gigabyte
GHz	Gigahertz
s	Second
Hz	Hertz
ns	Nanosecond

ABBREVIATIONS

CAS	Control Actuation System
CPU	Central Processing Unit
DC	Direct Current
DLL	Dynamic Link Library
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
HIL	Hardware In The Loop
IDE	Integrated Development Environment
MBD	Model Based Design
PWM	Pulse-Width Modulation
RCP	Rapid Control Prototype

CHAPTER 1

INTRODUCTION

Co-simulation allows participating domains to develop with the most suitable simulation tools during modelling subsystems. In addition, co-simulation also encourages re-usability of systems. Co-simulation of dynamic systems allows detecting problems before integration and guiding developers while making design choices. Traditional co-simulation environments does not require meeting real-time requirements. To enhance further development to co-simulation real-time components could be introduced to system. These components be either hardware or model which meets real-time requirements. In order to provide a co-simulation environment, following problems need to be solved: integration of different simulation tools, accurate data exchange, coupling of systems with different dynamic behaviour, and guarantee of the accuracy of the simulation. One needs to take into consideration round trip time and noisy sensor signals. In order to provide stability and assure reliability in the system round trip time should be kept small and noisy sensors should be taken into consideration. [1]

The essence of the difference between real time and non-real time simulations is that real-time systems meeting so called hard real-time requirements. For example, guaranteed response time or deterministic run-time behavior. Non real-time systems generally does not meet those requirements and their run time behavior can be faster or slower. Faster systems are also called quasi real-time systems. When a real-time simulation is being performed, one of the main problems is the dead times when communication is taking place between models or processing and computation is performed inside the model. Even though these delays can be ignored according to the needs of the co-simulation scenario and environment, when simulating a case from control field these dead times can cause oscillations and result in erroneous results. As a result of this problem, delay analysis should be performed on real-time frameworks.

1.1 Motivation

Co-simulation environments allows contributing fields to develop models using the most suitable simulation tools. Models developed using different simulation tools can be integrated into the same co-simulation environment. Co-simulation method provides the opportunity to reuse models. Even though co-simulation has benefits, there are problems which need to be solved by the developer. These problems can be sorted as:

- integration of models which are developed using different simulation tools,
- accurate data exchange between models,
- coupling of systems with different dynamic properties
- accuracy of overall simulation [2]

Integration of models which are developed using different simulation tools such as Matlab, Simulink, Dymola is one of the main struggles. To provide a solution to this problem which addresses common interface aspect is Functional Mock-up Interface(FMI) standard. This standard can be applied to all models developed using different simulation tools. Models which conform to FMI standard are called Functional Mock-up Unit(FMU)s. FMI defines the interface functions. FMUs are expected to implement the required functions as described. Even though FMI brings a solution to interface description problem during co-simulation environment development, it does not dictate an architecture or orchestration standard. Allowing user to decide how to orchestrate FMUs provides flexibility to the design. Common practice for co-simulation applications is using a master-slave architecture. A model which acts as master coordinates the co-simulation scenario and performs data exchange between models. Master does not need to know about which code piece is running in the FMU which enables data and know-how protection when models are exchanged between different companies. Because of the nature of the FMI standard, data can be extracted from the model using getXXX(XXX = Real, Integer, Boolean, String) functions and fed to relevant model using setXXX(XXX = Real, Integer, Boolean, String) functions. This interface causes a bottleneck on the master algorithm when performing data exchange. In this thesis work, bottleneck on master component caused by data exchange between models through FMI Master is broken with an observer pattern application on variables exchanged between FMU Block components through FMI Master. Even though this framework is developed to break the dependencies between model and the FMI Master, to orchestrate models one always needs to call doStep method on models causing to keep master-slave architecture in order to perform doStep command.

1.2 Scope of the Thesis

Aim of this thesis is to provide a co-simulation framework which helps to ease co-simulation component integration and provides a foundation for real-time co-simulation environment. Models which are developed in different simulation environments conforming to FMI standard and hardware components which may communicate through serial interface such as RS485 can be integrated to a co-simulation environment using the proposed framework. The framework is a real-time hardware-in-the loop co-simulation tool, developed in C++ language to introduce object orientation to FMI concept. Developed framework is tested on QNX Neutrino operating system and delay introduced by object orientation and framework is analysed. Data exchange system provided by framework eases co-simulation environment development. To realize a simulation user provides data exchange information between components, simulation duration, termination condition, period and initial values of each model through a format specified xml file. Data exchange will be handled by the framework so that user does not need to hard-code the data exchange on the framework. If any changes occur in the variables, user may provide the changes to the platform by changing the xml file without being obliged to change the compiled code.

1.3 Limitations of the Study

The framework developed in this thesis work does not provide any opportunity regarding:

- a switch option for master algorithms from outside the framework: If simulation designer wants to change algorithms, then code of the master should be changed and framework should be recompiled.
- data interpolation: Data interpolation is a type of estimation method which constructs new data points within the range of discrete set of previously known data points obtained by sampling or experimentation in numerical analysis. This feature requires keeping the data sampled during the simulation so far and future sampling of the FMI models to estimate the points in between. Since this framework is intended for real-time use, sampling more than once can introduce overhead to falsify co-simulation results. Even though this framework does not examplify interpolation during a co-simulation scenario, it does not prevent it. Developer should analyze the overhead introduced by interpolation practice if needed.
- data extrapolation: Data extrapolation is a type of estimation method which constructs new data points beyond the range of discrete set of previously known data points. This feature requires to hold the previously sampled data during simulation. Even though this framework does not examplify extrapolation during a co-simulation scenario, it does not prevent it. Developer can perform extrapolation on the variables by keeping data log and running an extrapolation algorithm.

1.4 Outline

Outline of the thesis is given as below.

Chapter 2 gives background information about topics relevant to this thesis. Brief information about FMI Standard is presented. FMI for co-simulation is examined in detail and information is given about programming interface and logic flow. Basic concepts of model based design and material on necessity of real-time integration to co-simulation environment is presented. Classical approach to FMI Co-simulation is given. In addition subject-observer design pattern is represented.

Chapter 3 is about related work to this thesis. Jacobi and Gauss-Seidel orchestration algorithms are presented and why Jacobi algorithm is selected is discussed.

Chapter 4 is about designed framework in completion of this thesis. Framework architecture is explained in detail. Sequence and execution timeline related diagrams are provided.

Chapter 5 is about a case study which is developed using real-time framework for FMI co-simulation. First, components of co-simulation environment are explained and logic of simulation scenario is introduced. Framework overhead is discussed.

Chapter 6 provides a conclusion to this thesis. Also, gathers future work.

CHAPTER 2

BACKGROUND

This chapter presents background information gathered prior to this thesis.

2.1 Introduction

In this chapter background information collected prior to this thesis is presented. In background chapter FMI standard, FMI for co-simulation, real-time co-simulation, classical approach to FMI co-simulation and subject-observer design pattern concepts are investigated.

In FMI standard chapter firstly history of the standard and functional mock-up unit(FMU) concept is presented.

In FMI for co-simulation section, concept of co-simulation in FMI standard is explained in detail. Later, flow of FMI co-simulation is explained and programming interface of FMI co-simulation is explained in detail.

In real-time model based co-simulation section, benefits of model based design in co-simulation is explained.

2.2 FMI Standard

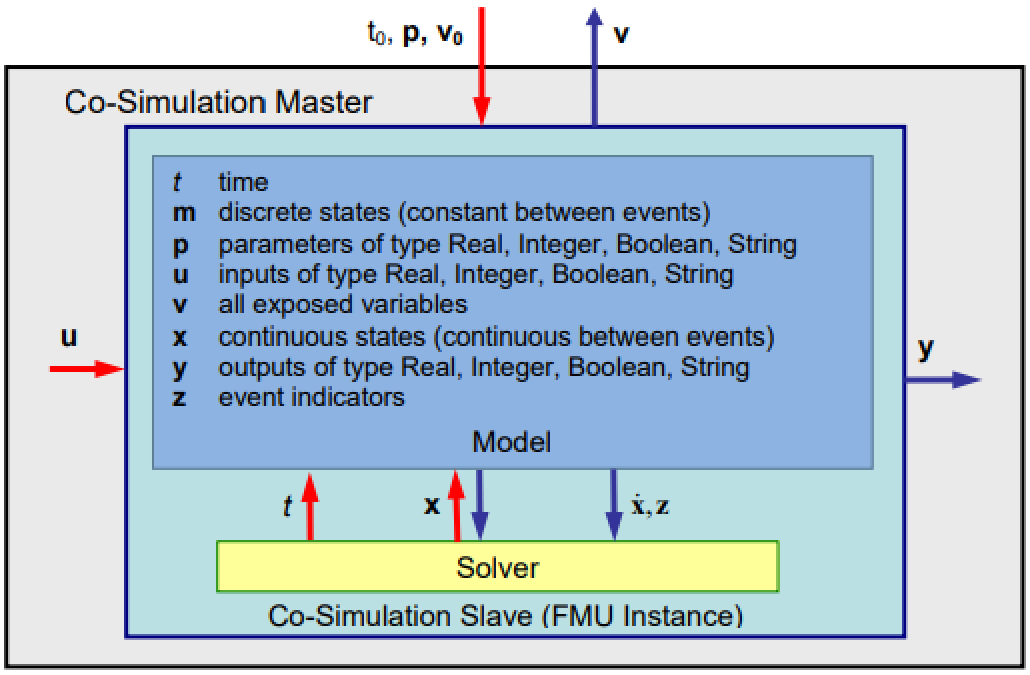
In this section, FMI standard will be introduced in detail. FMI is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of xml files and C-code in either compiled as DLL/shared libraries or source code. FMI was developed in a European project, MODELISAR, that was focused on improving the design of systems and of embedded software in vehicles. The standard is now maintained and developed by the Modelica Association. Since its release, the standard has received a significant amount of attention among both tool vendors and users. The large number of tool vendors that have adopted the standard shows that there is a real and pressing need to be able to export and import dynamic system models between existing tools, and also to be able to develop custom simulation environments. The first version, FMI 1.0, was published in 2010. Daimler AG initiated the FMI development with the goal to improve the Exchange of simulation models between suppliers and OEMs(Original Equipment Manufacturer). As of today, development of the standard continues through the participation of 16 companies and research institutes. FMI 1.0 is supported by over 100 tools and is used by automotive and non-automotive organizations throughout Europe, Asia and North America. Examples include the commercial products Dymola and SIMPACK, as well as the open-source platform JModelica.org. FMI 2.0 was released on July 25th 2014. In FMI 2.0, FMI 1.0 Model Exchange and Co-Simulation standards have been merged, and many improvements have been incorporated. New features are usually optional. [3]

An FMU is a model without integrators, a runnable model with integrators, or a tool coupling interface. In short, FMU is a model, which complies FMI standard.

A schematic view of an FMU is shown in Figure 2.1. Blue arrows represent information provided to FMU. Red arrows represent information provided by FMU.

2.3 FMI for Co-simulation

FMI for Co-simulation is a standard interface for the model that contains its solver inside. In co-simulation, main intention is to provide an interface standard to coupling models, which can be developed in different simulation tools in a co-simulation environment. Data exchange in co-simulation is restricted to discrete communication points. Between communication points FMUs are solved independently from each other. Master algorithm controls Exchange of data between FMUs. FMI standard does not specify a certain master algorithm. General structure of a co-simulation



t_0, **p**, **v**$_0$
v
Co-Simulation Master
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
Model
u
y
t
x
$\dot{\mathbf{x}}, \mathbf{z}$
Solver
Co-Simulation Slave (FMU Instance)

Figure 2.1: Data Connection Between Environment and FMU [3]

FMU can be shown as Figure 2.2.

A co-simulation FMU contains its solver inside which allows FMU to be used as an independent component. To use a given FMU in co-simulation one should perform three steps: instantiate and initialize, run and terminate simulation. In the instantiation and initialization step, an instance of FMU is created and initialized. In this step, memory allocation needed by the FMU is performed and variables in the FMU are set to initial values. During simulation, master can use defined setter, getter and execution methods to run model. Setter methods allows master to communicate with models

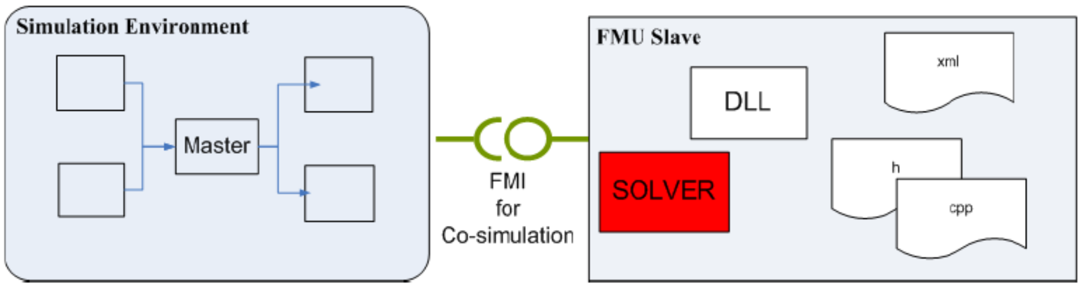


Figure 2.2: UML 2.0 State Machine for Co-Simulation FMU [3]

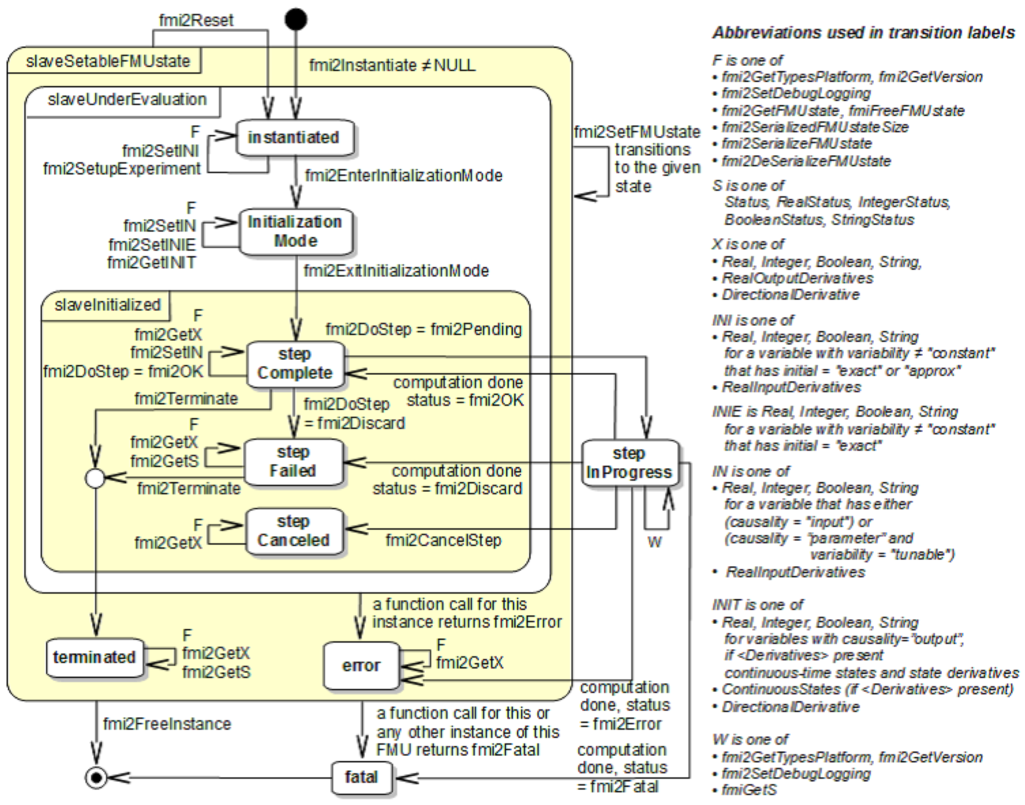

fmi2Reset
slaveSetableFMUstate
fmi2Instantiate ≠ NULL
slaveUnderEvaluation
F
fmi2SetINI
fmi2SetupExperiment
instantiated
fmi2EnterInitializationMode
F
fmi2SetIN
fmi2SetINIE
fmi2GetINIT
Initialization Mode
fmi2ExitInitializationMode
slaveInitialized
F
fmi2GetX
fmi2SetIN
fmi2DoStep = fmi2OK
step Complete
fmi2DoStep = fmi2Pending
computation done status = fmi2OK
fmi2Terminate
fmi2DoStep = fmi2Discard
F
fmi2GetX
fmi2GetS
step Failed
computation done status = fmi2Discard
fmi2Terminate
F
fmi2GetX
step Canceled
fmi2CancelStep
step In Progress
W
fmi2SetFMUstate transitions to the given state
a function call for this instance returns fmi2Error
terminated
F
fmi2GetX
fmi2GetS
error
F
fmi2GetX
computation done, status = fmi2Error
fmi2FreeInstance
a function call for this or any other instance of this FMU returns fmi2Fatal
fatal
computation done, status = fmi2Fatal
Abbreviations used in transition labels
F is one of
• fmi2GetTypesPlatform, fmi2GetVersion
• fmi2SetDebugLogging
• fmi2GetFMUstate, fmiFreeFMUstate
• fmi2SerializedFMUstateSize
• fmi2SerializeFMUstate
• fmi2DeSerializeFMUstate
S is one of
Status, RealStatus, IntegerStatus, BooleanStatus, StringStatus
X is one of
• Real, Integer, Boolean, String,
• RealOutputDerivatives
• DirectionalDerivative
INI is one of
• Real, Integer, Boolean, String for a variable with variability ≠ "constant" that has initial = "exact" or "approx"
• RealInputDerivatives
INIE is Real, Integer, Boolean, String for a variable with variability ≠ "constant" that has initial = "exact"
IN is one of
• Real, Integer, Boolean, String for a variable that has either (causality = "input") or (causality = "parameter" and variability = "tunable")
• RealInputDerivatives
INIT is one of
• Real, Integer, Boolean, String for variables with causality="output", if <Derivatives> present continuous-time states and state derivatives
• ContinuousStates (if <Derivatives> present)
• DirectionalDerivative
W is one of
• fmi2GetTypesPlatform, fmi2GetVersion
• fmi2SetDebugLogging
• fmiGetS

Figure 2.3: Co-simulation [3]

and provide data exchange between them. To solve model's differential equations, doStep method is defined by the FMI standard. The produced output parameters exchanged with other FMUs by using getter methods. In the running step firstly, input parameters are set by calling setter methods (*FMUSetX(X can be either Real, Integer, Boolean or String)*). Then *doStep()* method is called to solve the model's differential equations. By calling getter methods *(FMUGetX(X can be either Real, Integer, Boolean or String))*, output parameters of the solved model can be accessed by the caller. In the termination step, resources used by the FMU are freed.

2.4 Real-time Co-simulation

Model based design is a mathematical and graphical method of addressing problems associated with the design of complex systems. MBD is a methodology based on V diagram. Through models, engineers involved in design and development of a project can exchange knowledge in an efficient and organized manner. [4] In addition, reuse

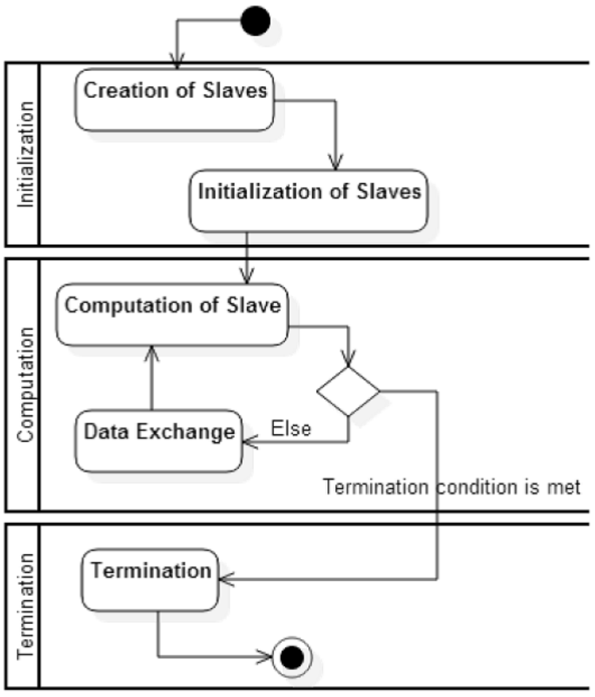


Figure 2.4: Co-simulation

of older designs is encouraged due to homogeneous design environment.

Automatic code generation features of commercial tools accelerates real-time simulation in MBD. By using Automatic Code Generator, a real-time simulator in other words a RCP(Rapid Control Prototype) can be implemented from a model with minimal effort. In contrast to offline simulation, using the prototype in HIL test bench testing can accelerate integration and verification testing. Due to involvement of test engineers in early stage of development process, design issues can be discovered earlier. In addition, prototyping may

- decrease development time,
- reduce cost of development cycle,
- reduce testing costs due to HIL test setup instead of physical setup,
- provide more repeatable test results due to unchanged dynamics of real-time simulators,
- provide alternative to expensive or risky tests.

During development of complex systems, integration of the components that com-

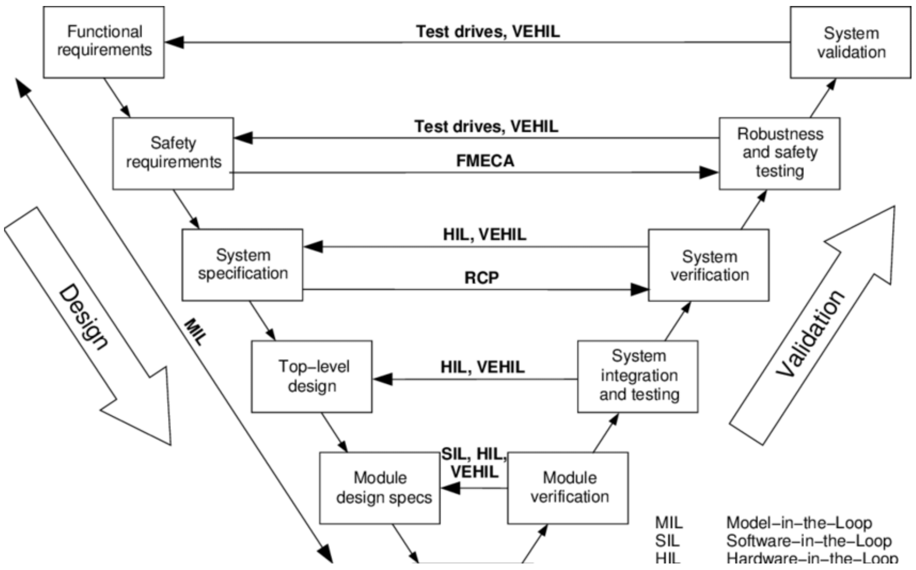


Figure 2.5: V Diagram for Embedded Design [5]

prise a system can be identified as major source of problems. [6,7] Finalizing a communication interface among different components of a large system which are being built in different teams inside the organization or even inter companies. Also, the features which did not have been discussed during meetings or covered in documents can cause assumptions on the product. Usually on integration process problems roots down to assumptions on other components to be made in early stages of development. [8,9] The lack of information can cause wrongly made assumptions which are harder to detect. As an example, speed regulator control algorithm can be developed using metric unit even though sensor is providing imperial units to control block. Despite challenges, co-simulation remains effective in reducing development time and improving quality as pointed out by the industrial partners contributed to the DESTECS project. [10–12]

Co-simulation is an integration technique to couple multiple simulators. Simulators often seen as black-box as in the FMI standard. As a result, advancement of states and time is completely hidden outside of the model and also not specified by the standard. Hidden advancement allows for specialized solvers to be used for the particular system which may lead to increase in performance and stable simulation.

2.5 Classical Approach to FMI Co-Simulation

In this section, classical approach to FMI co-simulation will be explained in detail. As previously visited in 2.3, when performing co-simulation, two basic groups of functions have to be realized:

- functions for the data exchange between subsystems
- functions for algorithmic issues to synchronize the simulation of all subsystems and to proceed in communication steps $tc_i \rightarrow tc_{i+1}$ from initial time $tc_0 := t_{start}$ to end time $tc_N := t_{stop}$

In FMI for Co-Simulation, both functions mentioned above are implemented in the co-simulation master. The data exchange between the slaves or subsystems is handled via the master node. In this architecture, there is no direct communication between the slaves. A special software tool can be used to implement master functionality. Also simulation environment can be generated in a nested form where in every level of hierarchy FMI for Co-simulation is applied. FMI for Co-simulation defines the interface routines between master and slave components.

During simulation, classical master algorithm stops at each communication point tc_i of all slaves, collects the outputs $y(tc_i)$ from each component or FMU, evaluates the subsystem inputs $u(tc_i)$, distributes these subsystem inputs to the slaves and continues the co-simulation with the next communication step $tc_i \rightarrow tc_{i+1} = tc_i + hc$ the with fixed communication step size hc. In each slave, an appropriate solver is used to integrate one of the subsystems for a given communication step $tc_i \rightarrow tc_{i+1}$. If there is any unknown input data, the most simple co-simulation algorithms approximate the subsystem inputs $u(t), (t > tc_i)$ by frozen data $u(tc_i)$ for $tc_i \leq t < tc_{i+1}$.

FMI for Co-Simulation supports the classical brute force approach as well as more sophisticated master algorithms. FMI for Co-Simulation is designed to support a very general class of master algorithms but it does not define the master algorithm itself. For slaves to be able to support more sophisticated master algorithms, inside the FMU's XML description there are capability flags which can be listed as:

- the ability to handle variable communication step sizes hcu

- the ability to repeat a rejected communication step $tc_j \rightarrow tc_i + i$ with reduced communication step size
- the ability to provide derivatives with respect to time of outputs to allow interpolation
- the ability to provide Jacobians

FMI for Co-Simulation is restricted to slaves with the following properties:

- All calculated values $v(t)$ are time dependent functions within an a priori defined time interval $t_{start} \leq t \leq t_{stop}$
- In general, simulations are carried out with increasing time. The current time t is running step by step from t_{start}, t_{stop}. The algorithm of the slave may have the property to be able to repeat the simulation of parts of $[t_{start}, t_{stop}]$ or the whole time interval $[t_{start}, t_{stop}]$.
- The slave can be given a time value $tc_i, t_{start} \leq tc_i \leq t_{stop}$
- The slave is able to interrupt the simulation when tc_i is reached
- During the interrupted simulation the individual solver of the slave can receive values for inputs $u(tc_i)$ and provide output values $y(tc_i)$.
- Whenever the simulation in a slave is interrupted, a new time value $tc_{i+l}, tc_i \leq tc_{i+1} \leq t_{stop}$ can be given to simulate the time subinterval $tc_i < t \leq tc_{i+1}$
- The subinterval length hc_i is the communication step size of the i^{th} communication step, $hc_i + tc_i = tc_{i+1}$. Communication step size can be more than or equal to 0. Usually step size does not take negative values. [13]

2.6 Subject-Observer Design Pattern

Subject observer is a design pattern which describes how to establish a structure with one subject to be observed by many observers. Pattern also includes the concept of notifying the observer when there's a change on the subject. In Figure 2.6 structure of subject observer pattern is presented. There are 4 participants:

- Subject: Provides an interface for observers to attach and detach. Has a data structure which holds the observers.
- Observer: Abstractly defines an update interface for objects which should be

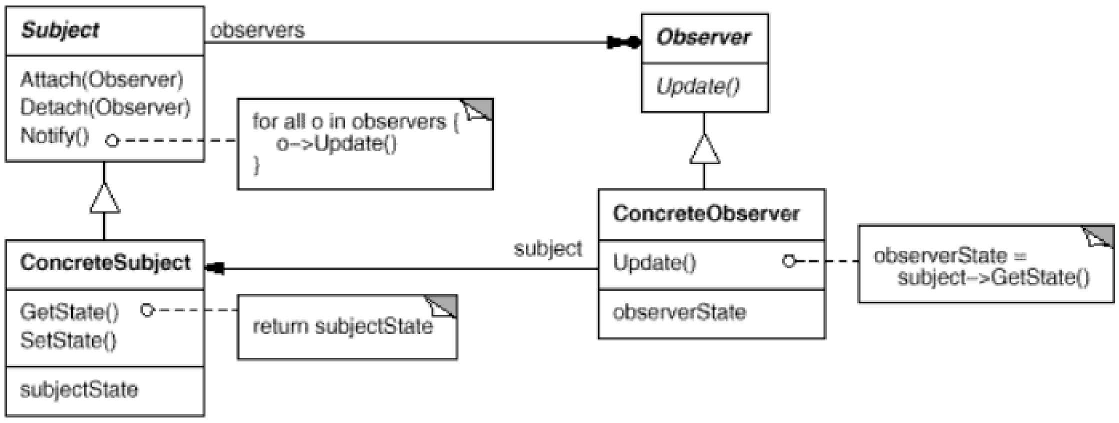


Figure 2.6: Subject Observer Structure [14]

notified by the subject.

- ConcreteSubject: Stores state of interest to ConcreteObserver objects. When a change occurs in the state, notifies the observers.
- ConcreteObserver: Holds reference to ConcreteSubject object. Stores the state which is consistent with the subject. Implements the observer update interface.

In Figure 2.7 sequence diagram for the observer pattern is displayed. In this scenario, observer updates the value of the subject which results in update of all observers attached to the subject which are aConcreteObserver and anotherConcreteObserver instances. When observers get notified, they're responsible for updating their state by getting the final state of the subject.

In Figure 2.8, a specification of a subject observer pattern in which the observers synchronize on an integer-valued property of the subject is given. Explanation of this specification in an informal way is: it is possible to identify a subject which has a notify method functions as updating the observer's state. All observers subscribed to the subject gets notified.

Advantages of the subject-observer pattern can be listed as below:

- abstract coupling between subject and observer: This feature enables subject and observer to belong in different layers of abstraction in the system.
- observers can be removed at any time: During run-time, observers can unsub-

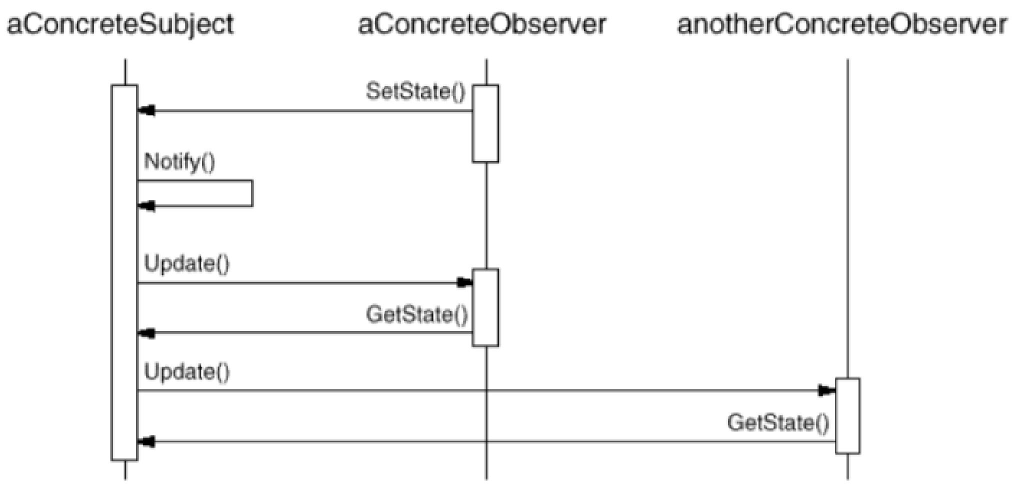
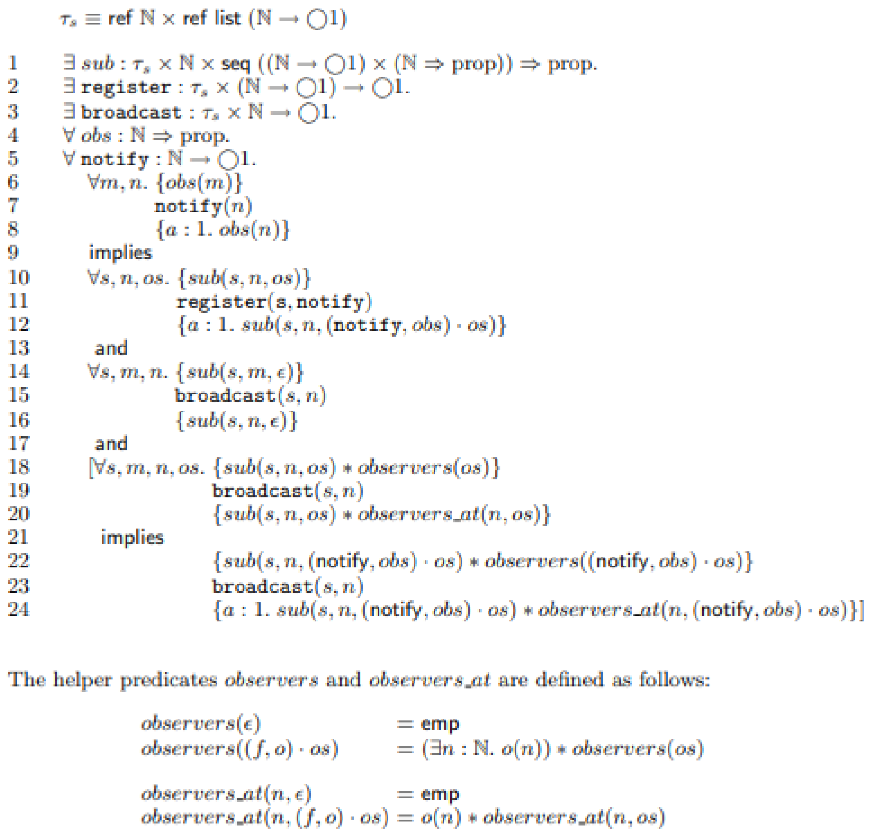


Figure 2.7: Sequence Diagram For Observer Pattern [14]



```
    $\tau_s \equiv \mathsf{ref}\ \mathbb{N} \times \mathsf{ref\ list}\ (\mathbb{N} \rightharpoonup \bigcirc 1)$

1   $\exists\, sub : \tau_s \times \mathbb{N} \times \mathsf{seq}\ ((\mathbb{N} \rightharpoonup \bigcirc 1) \times (\mathbb{N} \Rightarrow \mathrm{prop})) \Rightarrow \mathrm{prop}.$
2   $\exists\, \mathtt{register} : \tau_s \times (\mathbb{N} \rightharpoonup \bigcirc 1) \rightharpoonup \bigcirc 1.$
3   $\exists\, \mathtt{broadcast} : \tau_s \times \mathbb{N} \rightharpoonup \bigcirc 1.$
4   $\forall\, obs : \mathbb{N} \Rightarrow \mathrm{prop}.$
5   $\forall\, \mathtt{notify} : \mathbb{N} \rightharpoonup \bigcirc 1.$
6     $\forall m, n.\ \{obs(m)\}$
7             $\mathtt{notify}(n)$
8             $\{a : 1.\ obs(n)\}$
9     implies
10    $\forall s, n, os.\ \{sub(s, n, os)\}$
11                $\mathtt{register}(\mathtt{s}, \mathtt{notify})$
12                $\{a : 1.\ sub(s, n, (\mathtt{notify}, obs) \cdot os)\}$
13     and
14    $\forall s, m, n.\ \{sub(s, m, \epsilon)\}$
15                $\mathtt{broadcast}(s, n)$
16                $\{sub(s, n, \epsilon)\}$
17     and
18    $[\forall s, m, n, os.\ \{sub(s, n, os) * observers(os)\}$
19                    $\mathtt{broadcast}(s, n)$
20                    $\{sub(s, n, os) * observers\_at(n, os)\}$
21      implies
22                    $\{sub(s, n, (\mathsf{notify}, obs) \cdot os) * observers((\mathsf{notify}, obs) \cdot os)\}$
23                    $\mathtt{broadcast}(s, n)$
24                    $\{a : 1.\ sub(s, n, (\mathsf{notify}, obs) \cdot os) * observers\_at(n, (\mathsf{notify}, obs) \cdot os)\}]$
```

The helper predicates *observers* and *observers_at* are defined as follows:

$$
\begin{aligned}
observers(\epsilon) &= \mathsf{emp} \\
observers((f, o) \cdot os) &= (\exists n : \mathbb{N}.\ o(n)) * observers(os) \\
\\
observers\_at(n, \epsilon) &= \mathsf{emp} \\
observers\_at(n, (f, o) \cdot os) &= o(n) * observers\_at(n, os)
\end{aligned}
$$

Figure 2.8: Simple Subject Observer Specification [15]

scribe without causing any changes on the subject.

- reuse: Subject and observer classes can be reused independently.

Disadvantages of the subject-observer design can be listed as:

- unnecessary complexity: If not correctly implemented observer pattern can raise complexity and lead undesirable performance problems.
- unexpected updates: If an interface is used where observers can call changes on the subject as examplified in 2.7 figure, cascading update can be triggered and tracing down the cause of update can be harder.
- memory leak: This problem is also known as 'Lapsed listener problem'. Due to reference of the observer being kept inside the subject, if unregisteration of observers are not completed correctly, memory leak can occur especially in garbage collected languages like Java. When observer's lifespan is over, it should be detached from the subject. If it is not detached, reference kept inside the subject keeps the observer alive causing memory leaks.

CHAPTER 3

RELATED WORK

In this chapter, two orchestration algorithms will be introduced and pros and cons of this algorithms will be discussed.

3.1 Introduction

The algorithm which processes the co-simulation scenario and coordinates the execution of the simulation units is called the orchestrator and also known as master or coordinator. In this section Jacobi and Gauss-Seidel orchestration algorithms will be introduced and explained in detail.

3.2 Orchestration Algorithms

3.2.1 Jacobi Algorithm

Jacobi algorithm is an orchestrator which assumes that every simulation unit is input delayed. Input delayed algorithms don't require subsystems to be sorted as in run-time schedule. There's no difference when it comes to running the models in order either Run Model1 - Run Model2 or Run Model1 - Run Model2. This allows developer to be free from the consequences of the input-output relationship between models. However, due to the fact that models can be output reacive, getOutput methods should be invoked carefully.

Pseudo code of the Jacobi algorithm is given in Figure 3.1.

```
Data: The stop time $T$, a communication step size $H$, a co-simulation scenario with unit references
      $D$, and the order $\sigma$ of their inputs.
$t := 0$;                                                    // Simulation time
// Initialize variables
for $w \in D$ do
    $\boldsymbol{uc}_{[w]} := \boldsymbol{y}_{[w]} := \mathbf{0}$;            // Current I/O variables.
end
while $t < T$ do
    // Compute outputs in order
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;
        $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);
    end
    for $w \in D$ do
        doStep($w, H, \boldsymbol{uc}_{[w]}$);      // Compute $\boldsymbol{x}_{[w]}(t+H)$ from $\boldsymbol{x}_{[w]}(t)$ and inputs.
    end
    $t := t + H$;                                            // Advance time
end
```

Figure 3.1: Pseudo Code for Jacobi Algorithm [16]

3.2.2 Gauss-Seidel Algorithm

```
Data: The stop time $T$, a communication step size $H$, a co-simulation scenario with unit references
      $D$, and their order $\sigma$.
$t := 0$;                                                    // Simulation time
// Initialize variables
for $w \in D$ do
    $\boldsymbol{uc}_{[w]} := \boldsymbol{y}_{[w]} := \mathbf{0}$;            // Current I/O variables.
    $\boldsymbol{up}_{[w]} := \mathbf{0}$;                        // Previous input variables.
end
// Compute initial outputs
for $j = 1, \ldots, |D|$ do
    $w := \sigma(j)$;
    $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;   // Compute input from set of sources.
    $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);                 // Compute output.
    $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$;
end
while $t < T$ do
    for $j = 1, \ldots, |D|$ do
        $w := \sigma(j)$;
        $\boldsymbol{uc}_{[w]} := C_w\left(\left\{\boldsymbol{y}_{[v]} | v \in S_{[w]}\right\}\right)$;
        doStep($w, H, \boldsymbol{uc}_{[w]}, \boldsymbol{up}_{[w]}$);   // Compute $\boldsymbol{x}_{[w]}(t+H)$ from $\boldsymbol{x}_{[w]}(t)$ and inputs.
        $\boldsymbol{y}_{[w]} :=$ getOutput($w, \boldsymbol{uc}_{[w]}$);
    end
    for $w \in D$ do
        $\boldsymbol{up}_{[w]} := \boldsymbol{uc}_{[w]}$;                       // Update previous input.
    end
    $t := t + H$;                                            // Advance time
end
```

Figure 3.2: Pseudo Code for Gauss-Seidel Algorithm [16]

Gauss Seidel orchestrator is an input/output reactive algorithm. Developer needs to have the information on which outputs are used to compute which inputs, and at which times. This is necessary to sort the execution time of the models. Orchestrator has to make sure that all the models that are dependent on the output of other models have the access to fresh data. At the i^{th} co-simulation step, a unit m must be executed after unit n if $n \in S_{[m]}$ and m is input/output reactive. In the Figure 3.2, pseudo code is given. This algorithm is constructed under the assumption that the units can always be sorted.

3.2.3 Difference Between Jacobi and Gauss-Seidel Orchestrators

Difference between Gauss Seidel and Jacobi algorithms are caused by the principle of input/output reactiveness of the orchestrators. Jacobi algorithm is input/output delayed whereas Gauss Seidel algorithm is not. Consequences to this difference can be listed as below:

- When using Jacobi algorithm there's no need to sort the simulation units when calling the doStep function. However getOutput functions need to be sorted because simulation units can be output reactive/delayed. When using Gauss Seidel algorithm developer always need to sort the calling sequence of simulation units in an order that every unit has the right input during the simulation.
- When using Jacobi algorithm developer does not need to keep track of the previous inputs to each unit.
- Interpolation techniques cannot be used with Jacobi algorithm which may result less accurate simulation.
- Jacobi algorithm allows parallelism in contrast to Gauss Seidel algorithm.

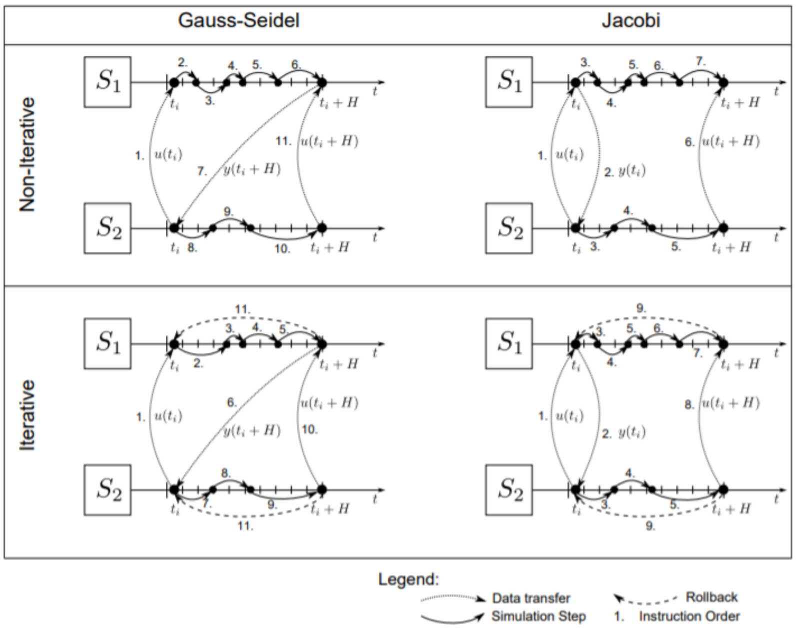


Figure 3.3: Gauss-Seidel & Jacobi Algorithms [16]

CHAPTER 4

REAL-TIME OBJECT-ORIENTED FRAMEWORK FOR FMI CO-SIMULATION

In this chapter, framework proposed to solve real-time co-simulation integration problem will be introduced and analyzed in terms of overhead caused by virtual function calls and simulation initialization.

4.1 Introduction

To execute a co-simulation scenario, developer needs to have knowledge further than the simulators and their data dependencies. These can be listed as:

- names and types of the variables used by the simulators
- time constraints of each simulator
- location of simulators
- coupling mechanism used to control the progress of the simulated time and move and adapt data among simulators

Dealing with data exchange when creating a co-simulation environment is an error-prone and time consuming task. To ease the coupling of co-simulation components a framework called Real-time Framework for FMI Co-simulation is created. User does not need to write and compile the code when there is a change in the co-simulation components or simulator interface variables if the master algorithm stays unaltered in co-simulation scenario. For this purpose framework can be compiled with different master algorithms and used later without any changes.

Using the framework requires input from user. These input variables are needed in order to successfully run a co-simulation. Developers usually have knowledge about data interface of each component and relationship between components. Although these information is needed, before the execution a co-simulation scenario, developer needs to have knowledge further than the simulators and their data dependencies. The information can be listed as:

- location of simulators
- names and types of the variables used by the simulators
- time constraints of each simulator
- coupling mechanism used to control the progress of the simulated time to move and adapt data among simulators

Gathering information on location of simulators is easily attainable. To collect the information about names and types of variables in the component, developers may use the help of FMI interface and extract the information from XML file of the simulation component. Determining the time constraints of the simulation components may not be always straightforward. Even though algorithm analysis' guidance can lead developers when predicting run-time behavior; to attain the time constraints of the component, developer may need to perform some tests. For example, to measure best and worst case step size of a component, developer needs to create a simulation scenario where shortest and longest paths in the algorithm logic in a step is called. By using this method developer can determine the step size of the component and make sure that this component returns a result even in the worst case scenario between the specified period. After collecting data regarding to co-simulation components separately, developer needs to decide on the master algorithm to couple and control the co-simulation scenario. Different component settings may require different algorithms. As an example, components composing the co-simulation scenario may have same or different periods.

Following sections in this chapter are to explain framework in detail. In 4.2 section, the aim of the framework is revisited and framework architecture is explained in detail.

4.2 Framework Overview

The aim of this thesis is to create a model integration and real-time co-simulation environment, which will automatically perform coupling between models during simulation according to information given by user. To create co-simulation environment without coding, user should specify following information in an xml file:

- models: This is the section where information about models are collected. Name, step size and initial values if desired.
  - step size of the model: default step-size of the model. This value cannot be variable step size.
  - initial values of model: values to be initialized in the model before co-simulation
- data dependencies between models: information about two models to be coupled should be presented here as their names specified in models part and names of the variables in each model. The reason why two variable names are present is variables to be coupled can have different names in corresponding models.
- termination condition: This can be either time constraint or value constraint. Time constraint stands for total simulation time of co-simulation. When time reaches to specified value, co-simulation terminates. Value constraints can be defined on models as observers. When the variable reaches desired value, co-simulation is terminated.

Schedule created according to xml file will be treated as if it meets the real-time requirements and co-simulation, will be run in a real-time environment. User should guarantee the performance requirement for each model and create simulation xml file accordingly. Creating a schedule, which meets real-time requirements by the master node, is not a part of this thesis. Data exchange between models will be performed automatically after each doStep execution by using observer pattern on variables. Each variable in an FMUBlock will be observable by other FMUBlocks if there is a data dependency between them. Observers will be attached to variables in order to be notified after a step. By this structure Master algorithm does not have handle the data exchange FMUs will perform it themselves.

Framework has a switch option for logging. Logging provides information about time and simulation details. By analyzing log data user can conclude the best and worst case step size of the model. Also, stream of data can reveal in which time interval which model has which data and best-worst step size of model. Using log data, user can decide whether simulation is performed according to hard real-time requirements.

4.2.1 Architecture In Classical Approach

In this section, architecture of the classical approach will be tackled and gains of the application of observer pattern and abstraction on the FMI Blocks will be investigated.

In Section 2.5, classical approach to FMI co-simulation is discussed. According to this, class diagram of a classical FMI co-simulation can be pictured as Figure 4.1, where FMU Blocks of different versions of FMI standard are connected with the FMI Master with one to many and bidirectional relationship. This results in FMI Master knowing all the FMI interfaces, keeping all the dependency relationships, all of the variables. When a change occurs in the simulation, FMI Master code needs to be altered and since all of the classes are dependant on it everything on the simulation should be tested again to verify changes on the FMI Master did not affected them. Also, if there's an addition to the FMI Interface, Master code should be written again. This results in a chunky FMI Master code which is hard to manage. To conclude there are 3 main problems with this approach:

- bottleneck on the FMI Master caused by holding all the information on inputs and outputs of the simulation components. Also, FMI Master performs the data exchange among the simulators.
- changes needed on the FMI Master when a new interface is available.
- changes on the FMI Master affects all nodes due to bidirectional relationship between slaves and the master.

To provide a solution to this structural problems, abstraction on FMU Blocks, application of subject-observer pattern on input output variables are implemented. By this way, independently of the simulation node's interface, FMI Master will be able to integrate this components. Also, FMI Master will not be responsible for the data exchange after each step which results in reducing the bottleneck on the FMI Master.

After the subject-observer pattern application on the IFMUBlock object, circular dependency between FMI Master, IFMUBlocks is eliminated. By this way, there's no need to keep the input/output variable's values inside the Master class, every IFMUBlock will hold their input/output values and other IFMUBlocks interested in the data will grab it from its source, not through FMI Master. In this scenario, when there's a change in variable sets on one of the FMU Blocks, since Master node is not altered as in classical approach, user does not need to test the Master code again to see if other components are affected by it. Only testing the new component will be sufficient.

In the following section framework will be explained in detail.

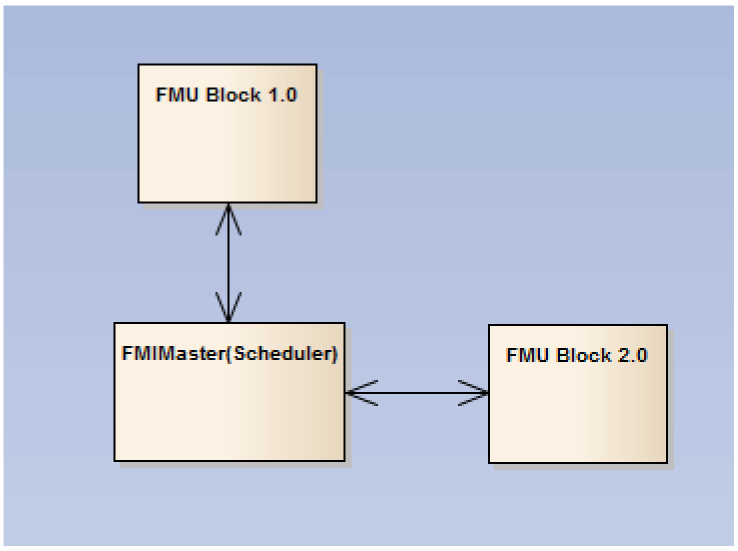


Figure 4.1: Circular Dependency In Classical Approach

4.2.2 Framework Architecture

In this section, framework architecture used in this thesis will be introduced.

Data exchange between models is performed through FMI Master in classical approach when performing a co-simulation. Models always rely on FMI Master to perform data exchange in this approach which creates a bottleneck on FMI Master when new data is produced by models after do step. To break dependency between models

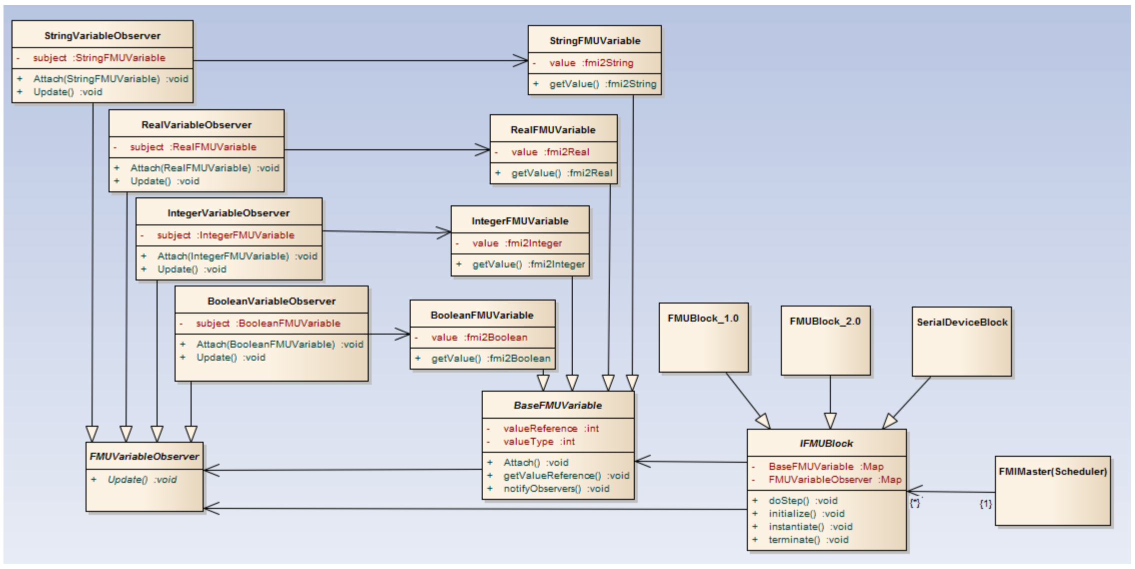


Figure 4.2: Class Diagram of Framework

and FMI Master and automatize data exchange, a framework is developed called Real Time Framework for FMI Co-Simulation. In this framework object oriented approach is used. Every model or hardware component is represented as IFMUBlock which is an abstraction for activity diagram of simulation sequence. This sequence consists of instantiation, initialization, step action and termination processes. Each block has a list of variables and observers. FMU Blocks related to each other are connected via variables and observers. Each variable has a list of observers which are attached to it. Observers are notified when a change occurs on the variable. Observer pattern also brings a solution to the bottleneck problem on FMI Master. In the instantiation and initialization step of the simulation:

- FMU Blocks which will take part in the simulation are created.
- FMU Variables are created and added to the list of the corresponding FMU Block
- Observers are created and attached to the corresponding variables and added to the observer list of the related FMU Block.

In the absence of FMI standard, components included in a co-simulation environment has their own interface between each other. This interface helps them to communicate and exchange data. There is no middle man which will exchange the data or orchestrate. Components usually run and exchange data within a period. In the classical approach FMI Master node is both responsible for data exchange and orchestration of the components. Even though different orchestration algorithms may need to be used and logging of the data exchange can be performed easier, it complicates the FMI Master and requires tightly coupling of the components. To bring a solution to data exchange delegation process, observer pattern is applied to the variables on each block.

During the simulation, when variables are updated after each step of FMU block, variable affected by the change will notify every observer which subscribed to the changes. By using this pattern, Master algorithm does not need to keep track of data exchange between models. Data exchange among models will occur automatically. By this way, FMI Master will only be responsible for orchestration of FMUs which will reduce the iteration time. In addition, this changes add value to the framework in terms of object oriented programming principles such as separation of concerns and encapsulation.

Separation of concerns property is provided to the framework by breaking the dependency on FMI Master. Components composing a co-simulation scenario are less tightly coupled. When a new FMI standard release is available or a new communication standard is introduced to the framework, developers will not be responsible for alteration on the FMI Master. This property will ease the process of troubleshooting and make testing easier. Framework prior to the change will be able to function as if no addition is made on the standard or communication blocks. Also, this allows developer to develop and perform unit tests on the latest classes without the need of testing the whole framework. If the latest classes are developed appropriately to the functional requirements and tested accordingly, whole framework does not need to be tested again.

Different orchestration algorithms can be used to co-simulate FMUs. It is up to developer to decide between algorithms. In this paper Jacobi and Gauss-Seidel orchestration algorithms presented in 3.2 and Jacobi algorithm is chosen. The reason why Jacobi algorithm is selected is due to the fact that user does not need to keep track of the sorting on the input/output reactive models. This would require user of this framework to sort the simulation units in a sense that the models which needs to run

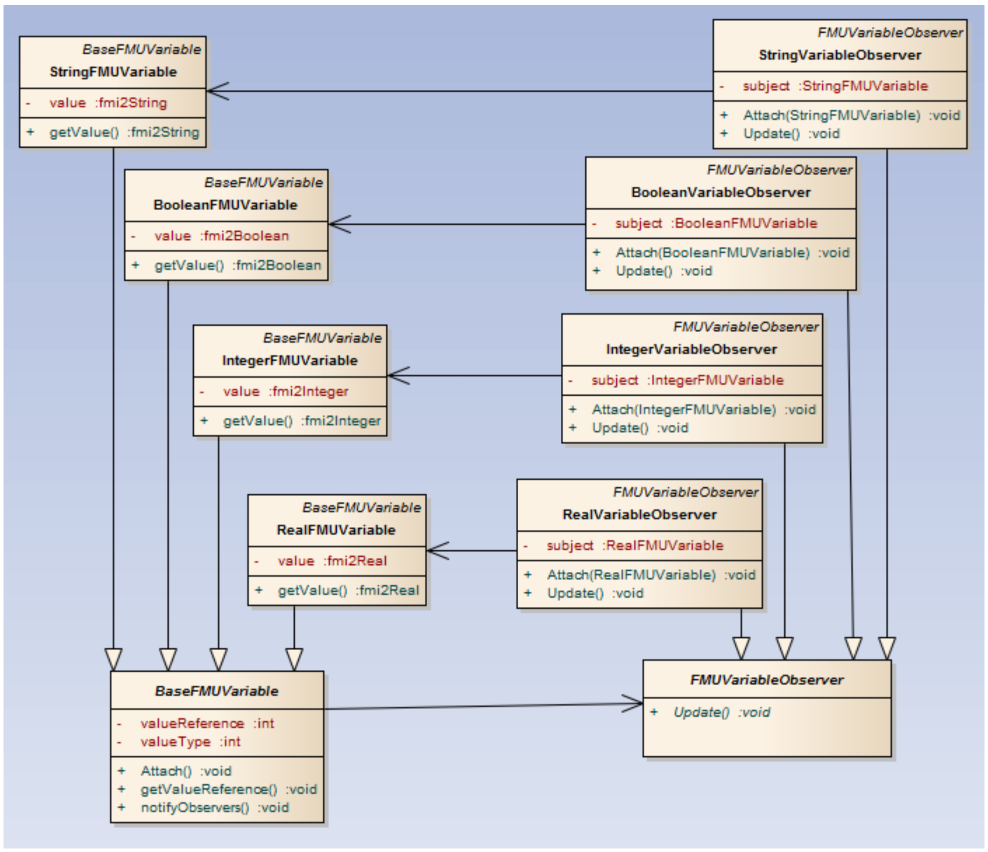


Figure 4.3: Observer Pattern on FMUs

beforehand is listed in the upper part of the simulation.xml file. Sorting can be performed in the xml file unless there's a cyclic dependency between models. In my opinion Gauss Seidel algorithm cannot be generalized as much as Jacobi algorithm, because of this reasoning Jacobi algorithm is chosen as orchestration algorithm in this framework. Scheduling will take place according to step size information provided by the user. Master will not order doStep function multiple times in order to reach a stable point for the outputs of the FMU. If a step returns a result other than fmi2Ok, the simulation will be terminated. Even though yhese features are not enabled in this study, framework can be extended to qualify these features.

4.2.2.1 Class Diagram In Detail

In this section, observer pattern applied on the variables of the FMU blocks will be explained in detail. To break the circular dependency on the classical approach between FMI Master and the blocks observer pattern is introduced to framework. In the class

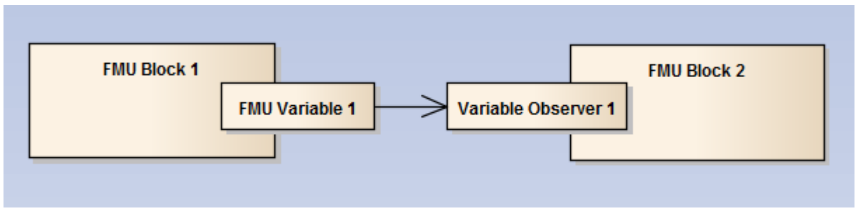


Figure 4.4: Observer Pattern Illustration

diagram 4.3 it can be observed that the BaseFMUVariable holds the value reference which is the index of the variable in FMI standard and value type. According to value type on the initialization process corresponding concrete type is created. Concrete types can be either one of FMI defined types: RealFMUVariable, IntegerFMUVariable, BooleanFMUVariable and StringFMUVariable. Data is held in those classes and they're only linked to the corresponding type's observer class. IntegerFMUVariable is only known by IntegerFMUVariableObserver. By this change, dependency arrow between FMI Master and FMU Blocks are only one way, not bi-directional. In addition, as illustrated in 4.4 in constrast to FMU variables not having a connection between them, they have links to the variables which are in their interest. Using this methodology, FMI Master does not need to regulate the data exchange between blocks during the whole simulation, on the initialization level all the variables and observers hence the IFMUBlocks are linked to perform data exchange when a change occurs on the variable and received by the dependent block right before step action. To sum up, FMI Master does not need to keep track of each variable and their final value, every block receives the latest data from the corresponding block itself.

4.2.2.2 Serial Communication

To handle the serial communication a class called SerialDeviceBlock is created. SerialDeviceBlock class inherits from IFMUBlock. For each serial device with RS485 communication interface this class should be used as a parent. Classes inheriting SerialFMUBlock should implement the message interface by adding each BaseFMUVariable object. On run time parsed xml file can link observers to this values. Also, observers from SerialDeviceBlock to the other IFMUBlocks can be added on the run time by specifying on the xml file. Only FMU variables should be hard coded to this class.

4.3 Co-simulation Environment Generation Using XML Files

The aim of this thesis is to create a model integration and real-time co-simulation environment, which will automatically perform coupling between components during simulation according to information given by user. To create co-simulation environment without coding, user should specify following information in an xml file:

- block: This is the section where information about models are collected. Name, step size and initial values if desired.
  - step size of the model: default step-size of the model. This value cannot be variable step size.
  - name: name of the FMU to find and extract variable from dynamic library.
  - version: version of the FMU. This can be either 1 or 2.
  - initial values of model: values to be initialized in the model before co-simulation.
- data dependencies between models: information about two models to be coupled should be presented here as their names specified in models part and names of the variables in each model. The reason why two variable names are present is variables to be coupled can have different names in corresponding models.
- Termination condition: This can be either time constraint or value constraint. Time constraint stands for total simulation time of co-simulation. When time reaches to specified value, co-simulation terminates. Value constraints can be defined on models as observers. When the variable reaches desired value, co-simulation is terminated.

Schedule created according to xml file will be treated as if it meets the real-time requirements. Framework does not guarantee component's response time it is user's responsibility to perform schedulability analysis. User should guarantee the performance requirement for each model and create simulation xml file accordingly. Creating a schedule, which meets real-time requirements by the master node, is not a part of this thesis. Data exchange between models will be performed automatically after each doStep execution by using observer pattern on variables. Each variable in an

FMUBlock will be observable by other FMUBlocks if there is a data dependency between them. Observers will be attached to variables in order to be notified after a step. By this structure Master algorithm does not have handle the data exchange FMUs will perform it themselves. Framework has a switch option for logging. Logging provides information about in which time interval which model has which data and best-worst step size of model. Using log data, user can decide whether simulation is performed according to hard real-time requirements. Overhead introduced to system by logging is analysed.

```
<Simulation>
        <Block>
                <Name>Guidance</Name>
                <Period>0</Period>
                <Version>1</Version>
                <InitialCondition>
                        <Parameter>
                                <Name>x</Name>
                                <Type>Real</Type>
                                <Value>10</Value>
                                <Index>1</Index>
                        </Parameter>
                </InitialCondition>
        </Block>
        <Block>
                <Name>Fuse</Name>
                <Period >50</Period>
                <Version>2</Version>
        </Block>
        <DataDependencies>
                <DataDependency>
                        <Type>Real</Type>
                        <Model1>Guidance</Model1>
                        <Index1>2</Index1>
                        <Model2>Fuse</Model2>
                        <Index2>1</Index2>
                </DataDependency>
        </DataDependencies>
        <TerminationCondition>
                <Duration>10<Duration>
        </TerminationCondition>
</Simulation>
```

4.4 Run Time Sequence Diagrams

To better understand the action in the background during run time, sequence diagrams on instantiation, initialization, step action, data exchange and termination are provided in this section. For simplicity, there are two FMU Blocks. FMU Block 1 is FMI 1.0 compliant and has one variable called FMU Variable 1. FMU Block 2 is FMI 2.0 compliant, dependent on FMU Block 1.

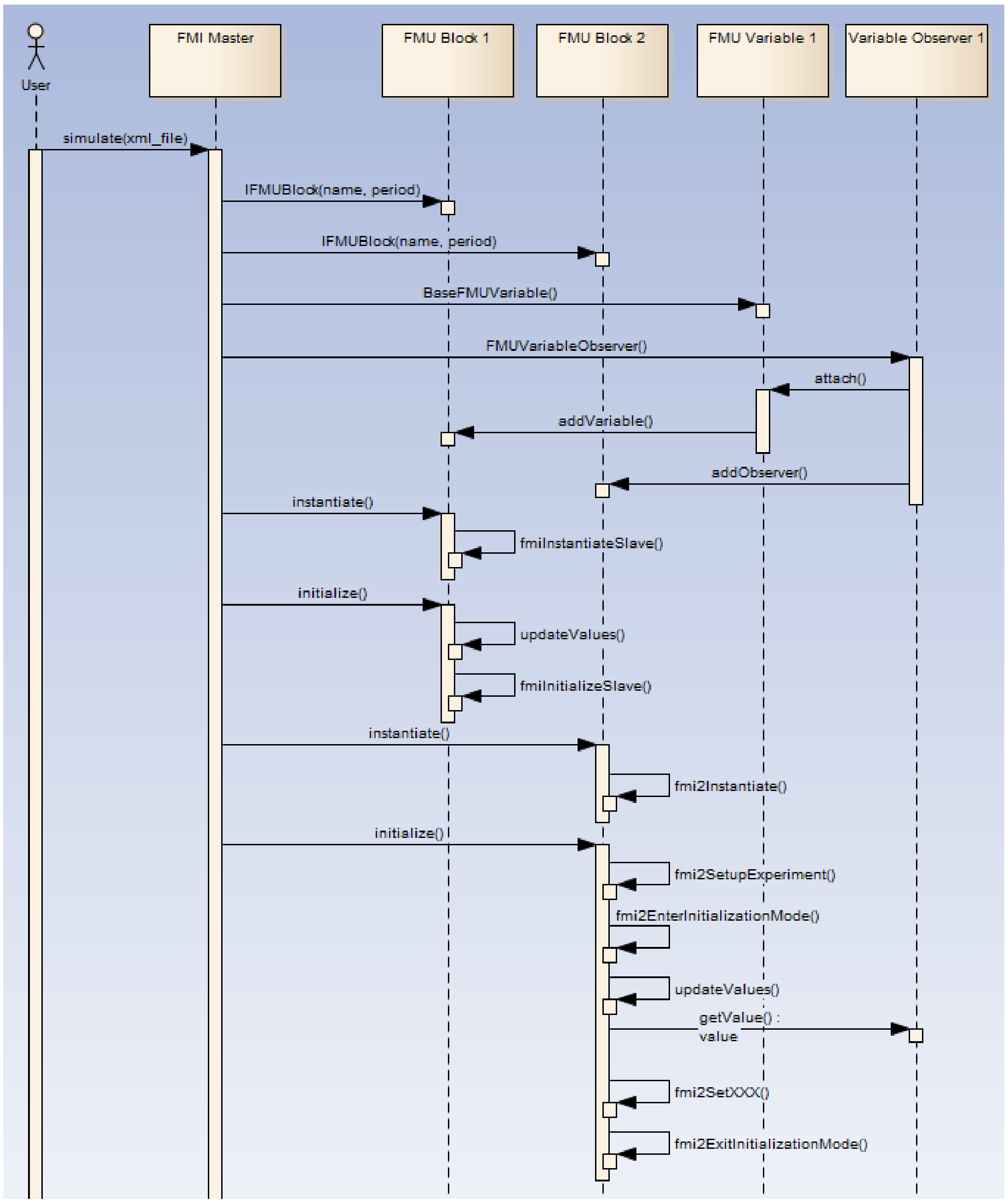


Figure 4.5: Initialization of Environment

In Figure 4.5, sequence diagram starts with simulation.xml input from user. FMI Master object takes the information given in the xml, parses it. In this XML there's information about the dependencies between models, model's name, period and version, duration of the simulation. According to version, model name and period information FMU Block objects are created. According to data dependency information given in the xml file first FMU Variable objects are created. If there are any initial values to the variable, the value is put inside the variable. After subject's creation observer objects are created and attached to the FMU Variable. On the instantiation .so(shared object) library is invoked for each FMU. In this architecture FMU Blocks are not implemented in the framework, they're called in the run-time. Using the FMI interface slaves are instantiated and initialized. On initialization step, if there are any initial values to the variables, setXXX(XXX= Real, Double, Boolean, String) methods are invoked.

In Figure 4.6, relationship between class objects when step action and data exchange during simulation is illustrated. FMI Master only orders doStep to slaves and FMI related methods and data exchange are performed in a hidden manner according to separation of concerns principle of the object oriented programming. Master does not contribute to data exchange between models. Before a step is prosecuted, if there is any observed variable in the FMU Block, values are pulled from the observed FMU Variable object. After completion of each step FMU Block's variables' are updated and observers are notified by them.

In Figure 4.7, termination process of the simulation is illustrated. First using the FMI interface and dynamic library handler FMU object is freed. Later, observers attached to the variables are detached. This step is cutial because if it is not performed memory leaks occur. Every observer should be detached then destroyed. Lastly, FMU variables and Variable Observers inside the FMU Block are destroyed.

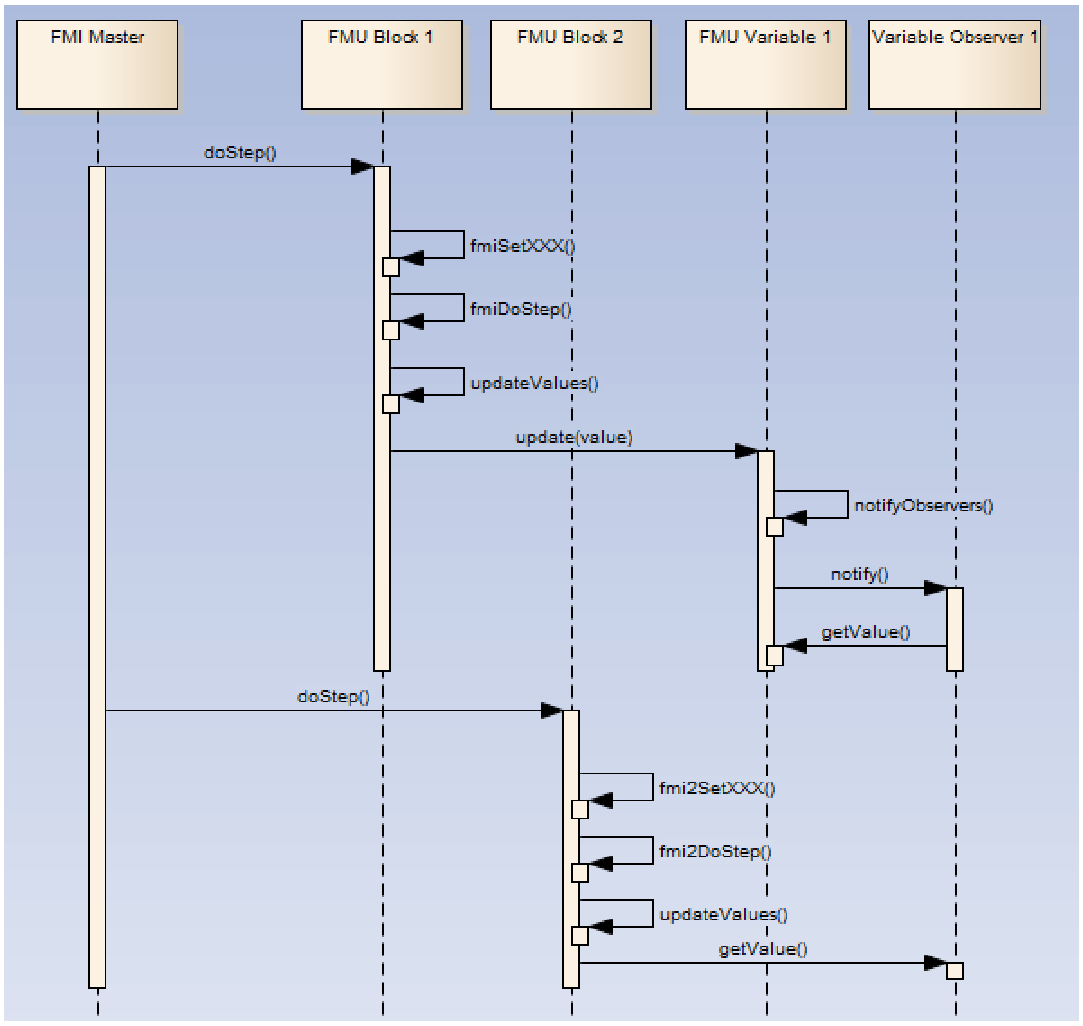


Figure 4.6: Simulation Time Sequence Diagram

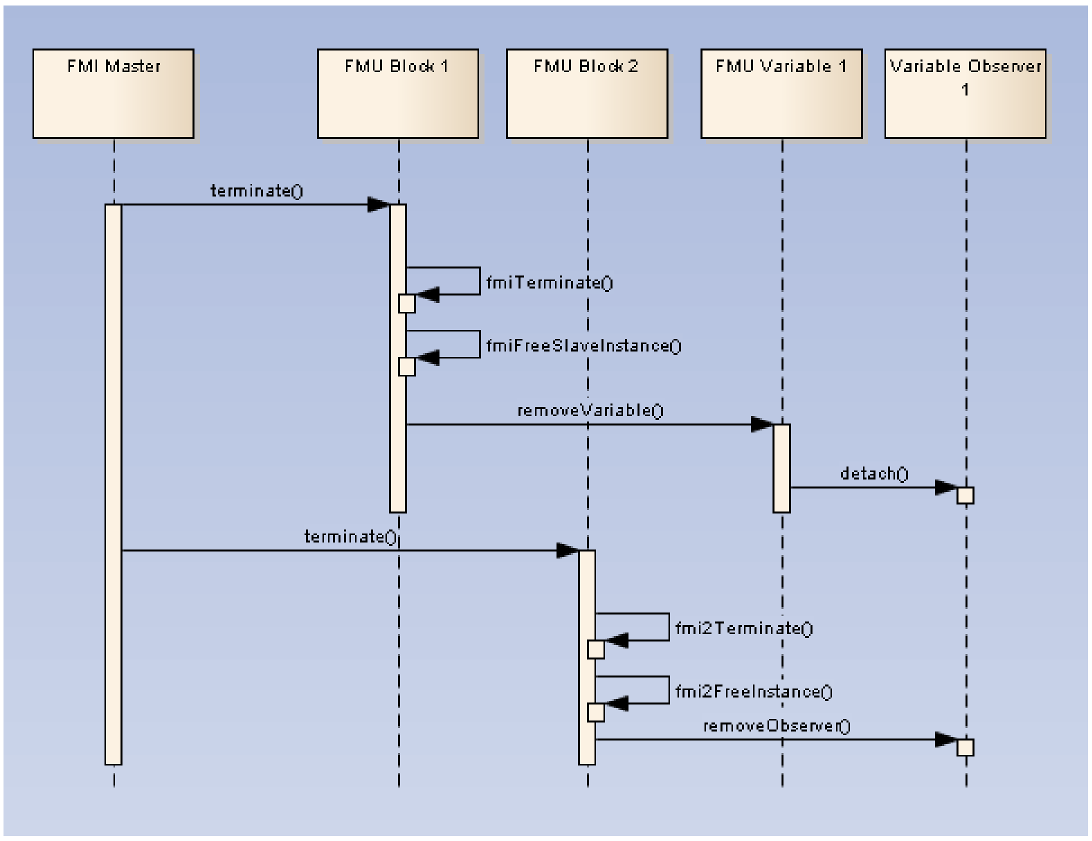


Figure 4.7: Termination Sequence Diagram

CHAPTER 5

CASE STUDY

This chapter provides a case study for the demonstration of real-time framework developed for this thesis. Framework is further analyzed in terms of functionality, determinism. As a co-simulation scenario a sine wave generator FMU and Control Actuation System is coupled. Co-simulation results are evaluated using a logging mechanism inside the FMI Master and deterministic behavior is verified using an oscillator. Case study is designed in order to allow user to understand the general concept better and provide an example to deterministic usage of the framework. Although framework does not guarantee meeting hard real-time requirements specified in input xml files, developers can use this framework regarding deterministic performance measures. Constructed case study is coded on QNX Momentics IDE and run on QNX operating system. In the following sections case study is explained in detail. Data dependency between hardware and software component is explained.

5.1 Case Study Overview

Real-time object oriented framework for FMI for co-simulation procures an integration environment for hardware and FMI compilant software components. In order to demonstrate the abilities of the framework, a co-simulation scenario is constructed. To represent hardware and software components, co-simulation environment incorporates a control actuation system(CAS) and a sine wave generator. This a scenario can be discussed as if we are the developer team of the CAS system, waiting for control algorithm to be developed by another team which will be developed according to FMI standard. Using the framework CAS system will be tested with a simple control scenario as a proof of concept on CAS is working according to requirements. In this

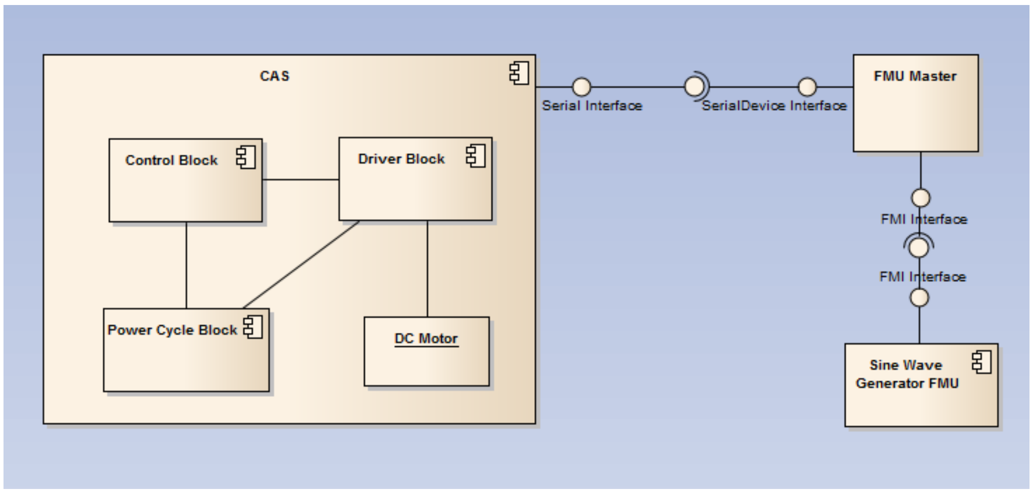


Figure 5.1: Component Overview

approach framework will guide the CAS developers to address and solve the bugs with a clear set of mind by eliminating the doubt about the control algorithm.

FMI Master orchestrates the simulation. CAS communicates through its interface using RS485 protocol. Sine Wave Generator uses FMI Interface. Figure 5.1 portrays a picture on communication interface. Details of the components are given below.

CAS is the hardware component in this scenario. In 5.1 sub components of the CAS can be observed. CAS comprises of 4 blocks which have functions as following:

- Control Block: This component manages communication with the outer world through RS485 network. Sampling of sensors such as temperature, current or potentiometer, control of DC motor position by sending pulse-width modulation(PWM) signals to driver block according to commands received via RS485 network and reporting the position changes to external system are the main responsibilities of this block.
- Driver Block: This component drives the DC motor to requested position according to PWM signals received from control block.
- Power Cycle Block: This component regulates the electrical power for the sensor components and processor by using the external source.
- DC Motor: The component which turns direct current electrical energy to mechanical energy.

Sine wave generator is a FMU which generates a floating point sine value between 0 and π according to simulation time variable. This FMU acts as a control algorithm for proof of concept purpose. Developers of the CAS control algorithm can replace the FMU with an algorithm more complex than a sine wave generator and observe the results. This property saves time when development of different components takes place in parallel within an organisation or in different companies. In example, think about a company which gave the development of the control algorithm job to a subcontractor and companies have agreed on using FMI interface to ease the integration process. In worst case scenario, subcontractor may face a problem on development of the algorithm which results in late delivery of the product. By using the framework's approach, instead of waiting until the algorithm is developed, CAS team can test their system without additional waiting time and troubleshoot any problems that may arise during integration process. By using FMI interface both companies save time because:

- deciding on a communication interface will not be an issue. Formal correspondence between companies in the process of deciding on an interface is time consuming.
- changes on the total variables will not be a concern on changing the interface if a variable is added or extracted. FMI interface is a fixed interface requires no changes.
- troubleshooting during integration process will be lessened due to early testing.

By using Real-time Framework for FMI Co-Simulation users will cut down on integration time because:

- in occurrence of a change in the interface variables, user will only need to change the simulation xml file.
- code of the FMU Master or any class does not need to be changed.

This case study is constructed to evaluate the following functionalities of Real-time Framework for FMI Co-Simulation:

- Automatized data exchange between slaves.
- Deterministic run-time behavior of co-simulation

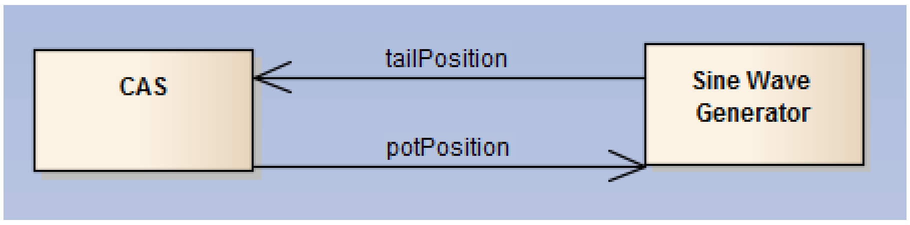


Figure 5.2: Data Exchange Between Components

- Communication between hardware and software components

In the following subsections data dependencies between components, creation of the Xml file for simulation environment generation and how real-time requirements of the scenario is handled are explained in detail.

5.1.1 Data Dependencies Between Models

In this section, data dependency between two models will be introduced. Figure 5.2 is a summary to the data exchange occurring between the components. CAS system requires a tail position command from the sine wave generator in order to rotate the wing in the desired position. Usually a control algorithm which commands the tail position decides according to the potentiometer position feedback acquired from the CAS. In our case even though sine wave generator does not calculate the tail position variable using potentiometer position, to realize the situation feedback variable is fed to the FMU as in real scenario. After the tests on CAS is over with one change in the simulation xml which is the name of the control algorithm, control algorithm can be replaced with the sine wave generator FMU and tests can continue. This is an example of decreasing deployment time property of the framework.

5.1.2 Creation of Xml File

In this section, using the information provided in 5.1.1 and following the instructions in 4.3 simulation.xml file will be generated as a sample. Data dependency graph is simple. There is a feedback loop between two components which can be summarized as CAS expects input angle from the sine wave generator FMU; sine wave generator FMU expects potentiometer position from CAS. To start with, one should start by

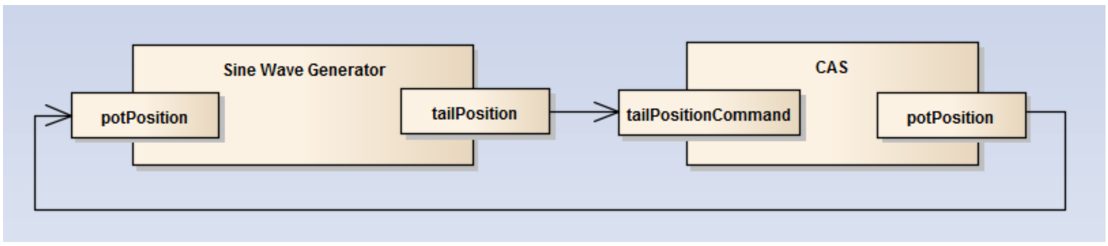


Figure 5.3: Data Dependency Between Models

creating an xml file which has the names, periods of the models and initial values to the variables such as following:

```
<Simulation>
        <Model>
                <Name>SineWaveGenerator</Name>
                <Period>100</Period>
                <InitialConditions>
                        <InitialCondition>
                            <!-- Index of tailPosition variable -->
                                <Index>1</Index>
                                <Type>Real</Type>
                                <Value>0</Value>
                        </InitialCondition>
                </InitialConditions>
        </Model>
        <SerialModel>
                <Name>CAS</Name>
                <Period>50</Period>
        </SerialModel>
</Simulation>
```

And then user should link the models to each other. Data dependencies between the models should be added to the xml. According to Figure 5.2 tail and potantiometer position values should be linked together. To achieve this user should add the following xml lines:

```
<DataDependencies>
        <DataDependency>
                <Type>Real</Type>
                <Model1>SineWaveGenerator</Model1>
                <Index>1</Index>
                <Model2>CAS</Model2>
                <Index>1</Index>
        </DataDependency>
        <DataDependency>
                <Type>Real</Type>
                <Model1>CAS</Model1>
                <Index>1</Index>
                <Model2>SineWaveGenerator</Model2>
                <Index>1</Index>
        </DataDependency>
</DataDependencies>
```

Finally, user should set the termination condition. User can determine this condition with different ways. For this scenario, duration is selected. Simulation is terminated after 10 seconds.

```
<Duration>10</Duration>
```

In the end, to run a simulation using the framework, a simulation.xml file is created as below:

```
<Simulation>
        <Model>
                <Name>SineWaveGenerator</Name>
                <Period>100</Period>
                <InitialConditions>
                        <InitialCondition>
                            <!-- Index of tailPosition variable -->
                                <Index>1</Index>
                                <Type>Real</Type>
                                <Value>0</Value>
                        </InitialCondition>
                </InitialConditions>
        </Model>
        <SerialModel>
                <Name>CAS</Name>
                <Period>50</Period>
        </SerialModel>
        <DataDependencies>
            <DataDependency>
                    <Type>Real</Type>
                    <Model1>SineWaveGenerator</Model1>
                    <Index>1</Index>
                    <Model2>CAS</Model2>
                <Index>1</Index>
            </DataDependency>
            <DataDependency>
                <Type>Real</Type>
                <Model1>CAS</Model1>
                <Index>1</Index>
                <Model2>SineWaveGenerator</Model2>
                <Index>1</Index>
            </DataDependency>
    </DataDependencies>
    <Duration>10</Duration>
</Simulation>
```

Table 5.1: Machine Configuration

Attribute	Value
CPU	Xeon E3 1275v3 3.5GHz 4 Core Processor
RAM	8 GB
OS	QNX 6.5.0 SP1
Momentics IDE	Version 7.0.3.v201804261557
Compiler	GNU Compiler Collection 4.4.2

5.2 Framework Overhead

Real-time Object Oriented Framework for FMI Co-simulation provides an object-oriented approach to integration of FMU slaves. To provide an insight to users of this framework to predict the overhead during simulation, this section is provided.

Developers considering to use the framework should run the performance tests to achieve more accurate results because processor used to run the simulation, compiler version, computer configuration, operating system differences can cause changes in the run-time behavior. In case study, simulations are run on the computer 5.1.

Table 5.2: Virtual Call Measurement

Number	With Virtual Call(ns)	Without Virtual Call(ns)
1	215966952	212967411
2	216966799	212967411
3	216966799	212967411
4	216966799	212967411
5	216966799	212967411
6	216966799	212967411
7	216966799	213967258
8	215966952	213967258
9	216966799	212967411
10	216966799	212967411

Framework can cause overhead in terms of the following aspects:

- Development language of the FMI interface is C. Language is changed to C++ to enable object-oriented development. Overhead can be caused by the following topics but these are not evaluated in scope of this thesis:
  - Language change has an impact on overhead which are evaluated in [17].
  - Object construction, object destruction, inheritance, dynamic method invocation.

Exception of the topics mentioned above, framework is investigated in order to measure the overhead caused by virtual calls. In this framework all of the variables are kept in the IFMUBlock using the template parent class BaseFMUVariable or FMUVariableObserver. This is inevitable overhead if developer wishes to use the simulation.xml file to realize the simulation rather than hard coding the variables on the framework. Also, simulation commands such as instantiate, initialize, doStep, terminate are given via IFMUBlock and overhead caused by this invocations cannot be eliminated.

Table 5.3: Virtual Call Measurement

Number	Difference(ns)	Difference/Virtual Call(ns)
1	2999541	0.02999541
2	3999388	0.03999388
3	3999388	0.03999388
4	3999388	0.03999388
5	3999388	0.03999388
6	3999388	0.03999388
7	3999388	0.03999388
8	1999694	0.01999694
9	3999388	0.03999388
10	3999388	0.03999388

An experiment is created to measure virtual function call overhead. This experiment consists of two classes: one parent and one child. Method is called directly from the instance of child class 10^8 times and measurements are taken. Also, the same method is called from the parent class 10^8 times and measurements are taken. Results are presented in the Table 5.2. This experiment is run on the machine which has the configurations specified in the Table 5.1.

Measurement is repeated 10 times with the same configuration and result for each run is given. The computation times was measured via the number of CPU cycles divided by the CPU cycle time.

Difference between the performance measurement of with and without virtual calls is given in Table 5.3. From this measurements it can be observed that at each virtual call an overhead is approximately 0.036994339 ns per virtual call.

In Table 5.4, virtual calls per class are represented. During the simulation because of the virtualization, IFMUBlock block delagates the instantiate, initialize, doStep and terminate functions to children classes: FMUBlock_1.0, FMUBlock_2.0 and SerialDeviceBlock. Each call of these functions from the FMI Master causes virtual function call. For instantiate, initialize and terminate functions this happens once.

For doStep function, it depends on the simulation time and the step size of the corresponding block. This number is represented as number N in the table. For example, if simulation is 10 seconds and every step is 1 second number N is 10. Before each doStep call on a model setInput functions are called. These functions are: for FMI 1.0, fmiSetReal, fmiSetDouble, fmiSetString and fmiSetBoolean; FMI 2.0, fmi2SetReal, fmi2SetDouble, fmi2SetString, fmi2SetBoolean. After each doStep function call, getOutput functions are called. For each variable v in the output set of a model, Update function will be called N times. If a model has M number of variables, M*N times Update function will be called.

Table 5.4: Number of Virtual Calls

Class	Virtual Function	Number Of Calls
IFMUBlock	other than doStep	1
IFMUBlock	doStep	N
FMUVariableObserver	Update	N

In summary, let's assume a co-simulation scenario with e number of simulation components and fixed step size. In initialization and instantiation initial values to the variables are assigned which results in $e * M$ times virtual calls. Each IFMUBlock calls instantiate, initialize and terminate functions once which results in $3 * e$ virtual calls. During simulation at each step $e * M$ times virtual Update function is called which results in $e * M * N$ virtual calls during simulation. Also each doStep call on IFMUBlock costs e virtual calls. In this scenario overhead can be calculated as:

$NumberofVirtualCalls = e * M * (N + 1) + 3e$

$Overhead = NumberofVirtualCalls * VirtualFunctionDuration/VirtualCall$

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this chapter, conclusion of this thesis and future work which can be implemented for hardware software component exchange purposes are presented.

6.1 Conclusion

The aim of this thesis is to present a model integration and real-time co-simulation environment, which will automatically perform coupling between components during simulation according to information provided to the framework. To realize this aim Real-time Framework for FMI Co-simulation is developed. In Chapter 4 framework class diagram and run time sequence diagrams are explained. By providing a case study in Chapter 5 usage of the framework is exampled. This framework is useful for integrating different FMI interfaces and hardware components which may or may not be integrated to framework using the FMI interface.

6.2 Future Work

In this section future work which can be constructed is presented. In Figure 6.1 there can be seen another class diagram using the observer pattern logic but with different FMU Block mentality. In this approach IFMUBlock is set to the latest FMI standard in order to incorporate only FMUs to the framework. In this approach FMI Master and Serial Device Block are also IFMUBlock. This provides pros and cons which can be listed as following:

- Advantages:

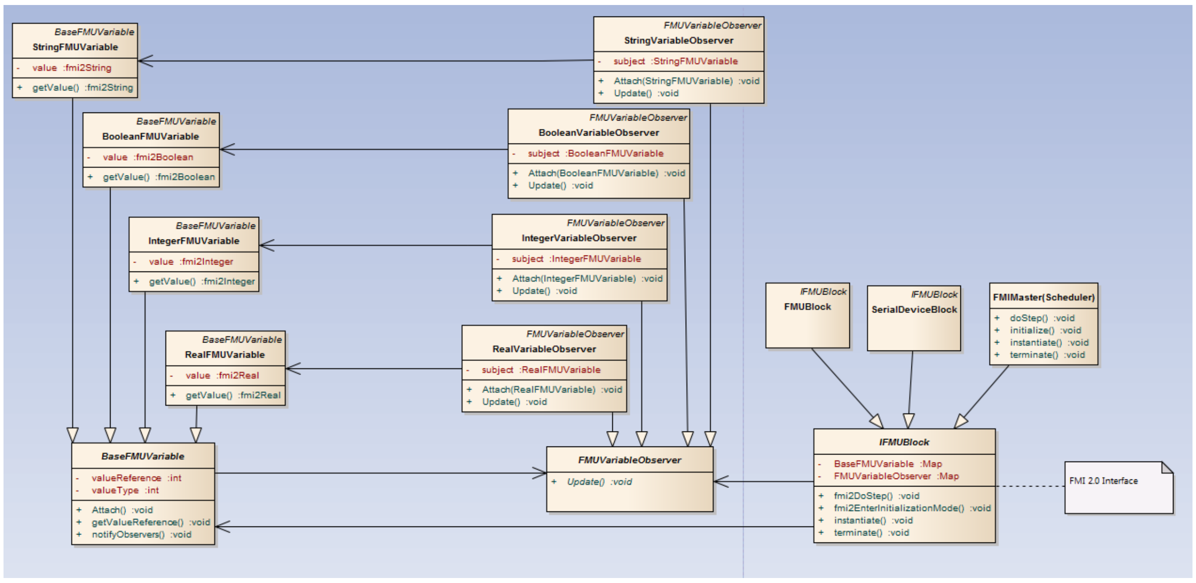


Figure 6.1: Future Work Class Diagram

- hardware and software components which perform the same task can be interchanged due to FMI 2.0 interface being the same in both. Previously hardware components did not need to comply with the FMI standard.
- FMI Masters can be used as a FMU component itself, not just an orchestrator.

- Disadvantages:
  - components with different versions of the FMI standard cannot be co-simulated with each other. There will be a need to write wrapper classes for different versions.
  - For every hardware component a wrapper class should be generated in order to comply with the FMI standard.

If the expectation from the framework is to exchange hardware and software components without any changes, than this architecture can be constructed.

REFERENCES

[1] G. Stettinger, J. Zehetner, M. Benedikt, and N. Thek, “Extending co-simulation to the real-time domain,” vol. 2, 04 2013.

[2] M. Benedikt, J. Zehetner, D. Watzenig, and J. Bernasch, “Moderne kopplungsmethoden - ist co-simulation beherrschbar?,” 11 2011. null ; Conference date: 08-11-2011 Through 09-11-2011.

[3] M. Consortium, “Functional mock-up interface for model exchange and co-simulation.”

[4] D. Auger, “Programmable hardware systems using model-based design,” pp. 1–12, 10 2008.

[5] M. Verhaegen, D. , O. Gietelink, J. Ploeg, and B. De Schutter, “Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations,” *Vehicle System Dynamics*, vol. 44, pp. 569–590, 08 2006.

[6] E. Lee, “Cyber physical systems: Design challenges,” pp. 363–369, 06 2008.

[7] T. Tomiyama, V. D’Amelio, R. J. Urbanic, and W. Elmaraghy, “Complexity of multi-disciplinary design,” *CIRP Annals - Manufacturing Technology*, vol. 56, pp. 185–188, 12 2007.

[8] O. Albayrak, H. Kurtoglu, and M. Biçakçi, “Incomplete software requirements and assumptions made by software engineers,” pp. 333–339, 12 2009.

[9] S. Uchitel and D. Yankelevich, “Enhancing architectural mismatch detection with assumptions,” in *Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000)*, pp. 138–146, April 2000.

[10] J. F. Broenink and Y. Ni, “Model-driven robot-software design using integrated models and co-simulation,” in *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 339–344, July 2012.

[11] U. Eliasson, R. Heldal, J. Lantz, and C. Berger, “Agile model-driven engineering in mechatronic systems - an industrial case study,” pp. 433–449, 09 2014.

[12] S.-A. Schneider, J. Frimberger, and M. Folie, “Significant reduction of validation efforts for dynamic light functions with fmi for multi-domain integration and test platform,” pp. 395–399, 03 2014.

[13] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. 11 2014.

[14] R. H. Erich Gamma, John Vlissides and R. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 10 1994.

[15] J. A. Neelakantan R. Krishnaswami and L. Birkedal, “Modular verification of the subject-observer pattern via higher-order separation logic,” *9th Workshop on Formal Techniques for Java-like Programs(FTfJP 2007)*, 2007.

[16] C. Gomes, C. Thule, P. G. Larsen, J. Denil, and H. Vangheluwe, “Co-simulation of continuous systems: A tutorial,” *ArXiv*, vol. abs/1809.08463, 2018.

[17] K. W. H. Hemant G. Rotithor and M. W. Davis, “Measurement and analysis of c and c++ performance,” *Digital Technical Journal*, vol. 10, no. 1, pp. 32–47, 1999.