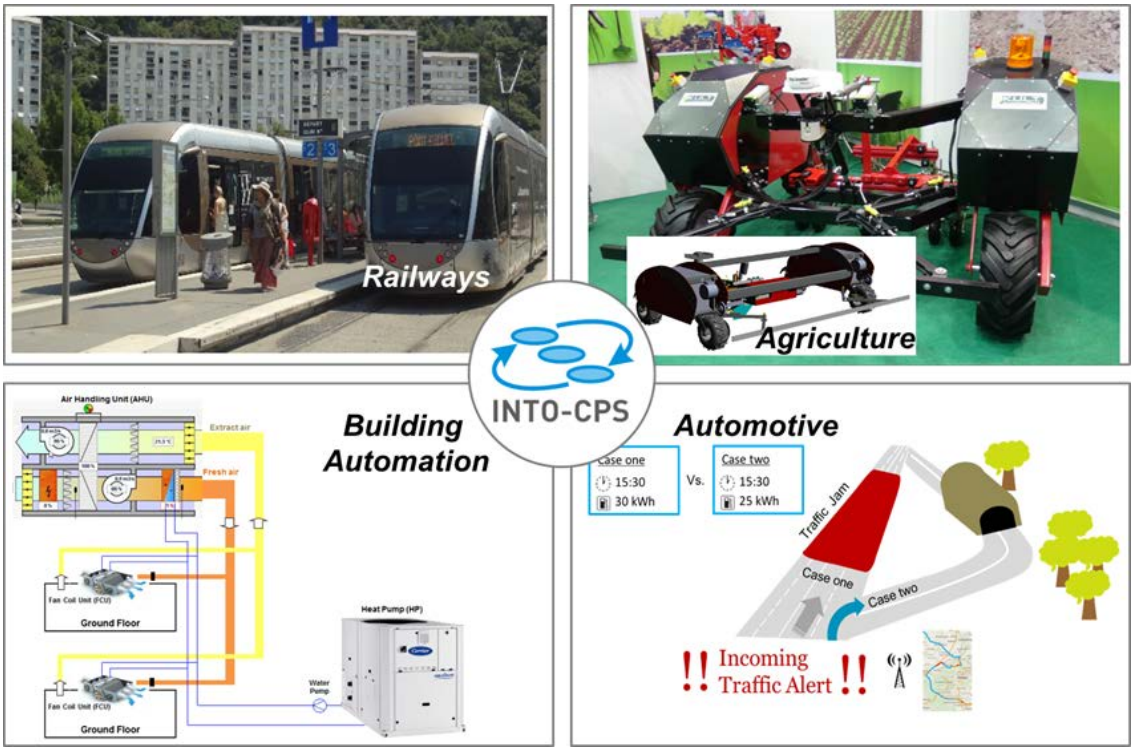
INVESTIGATING CONCURRENCY IN THE **CO-SIMULATION ORCHESTRATION ENGINE FOR INTO-CPS**

Electrical and Computer Engineering
Technical Report ECE-TR-26



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-26

Author: Casper Thule
Department of Engineering – Electrical and Computer Engineering, Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2016 Pages: 111
Editing completed: May 2016

Abstract: There is a tendency to expect, that taking advantage of multicore systems by using concurrency improves the performance of an application. To investigate if this is true, a case study was performed where different concurrency principles were applied to an existing application called the Co-Simulation Orchestration Engine (COE), which did not utilize concurrency. This was explored in the context of Co-Simulation using the Functional Mock-up Interface, as applications executing Co-Simulations should be performant to enable the use of increasingly complex models.
Co-Simulation can be useful in the development of Cyber-Physical Systems, as it can be used to simulate coupled technical systems or models and thereby examine the behavior of the systems.

The investigation was carried out by refactoring the COE to make it suitable for implementing concurrency by limiting the spawning of threads and synchronization between threads, along with maximizing the workload for each thread. Three different concurrency features were used in three different implementations: Parallel collections, futures, and actors, which were evaluated based on selected quality attributes. These implementations were tested against the non-refactored sequential COE and each other by performing different simulations using different models.

The case study showed, that concurrency can be used to increase the performance of the COE in some cases. Based on the analysis carried out in this thesis project, a set of guidelines were created to generalize the process of applying concurrency to an existing application.

Keywords: concurrency, functional programming, scala, co-simulation, cyber-physical systems, functional mock-up interface, and into-cps

Supervisor: Peter Gorm Larsen

Please cite as: Casper Thule, 2016. Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS. Department of Engineering, Aarhus University. Denmark. 111 pp. - Technical report ECE-TR-26

Cover image: INTO-CPS

ISSN: 2245-2087

INVESTIGATING CONCURRENCY IN THE CO-SIMULATION ORCHESTRATION ENGINE FOR INTO-CPS

Casper Thule
Aarhus University, Department of Engineering

Abstract

There is a tendency to expect, that taking advantage of multicore systems by using concurrency improves the performance of an application. To investigate if this is true, a case study was performed where different concurrency principles were applied to an existing application called the Co-Simulation Orchestration Engine (COE), which did not utilize concurrency. This was explored in the context of Co-Simulation using the Functional Mock-up Interface, as applications executing Co-Simulations should be performant to enable the use of increasingly complex models.

Co-Simulation can be useful in the development of Cyber-Physical Systems, as it can be used to simulate coupled technical systems or models and thereby examine the behavior of the systems.

The investigation was carried out by refactoring the COE to make it suitable for implementing concurrency by limiting the spawning of threads and synchronization between threads, along with maximizing the workload for each thread. Three different concurrency features were used in three different implementations: Parallel collections, futures, and actors, which were evaluated based on selected quality attributes. These implementations were tested against the non-refactored sequential COE and each other by performing different simulations using different models.

The case study showed, that concurrency can be used to increase the performance of the COE in some cases. Based on the analysis carried out in this thesis project, a set of guidelines were created to generalize the process of applying concurrency to an existing application.

Preface

As the finishing part of the Master's degree in Computer Engineering at Aarhus University, this Master's thesis is the final project of the education. It was conducted from August 24th 2015 to January 4th 2016 and accounts for 30 ECTS points.

This thesis project originates from the Integrated Tool Chain for Model-Based Design of CPSs (INTO-CPS) project [Larsen, 2015]. More specifically, the thesis project contributes to a part of the tool chain that concerns simulation and Co-Simulation.

I would like to thank my academic supervisor, Peter Gorm Larsen, for valuable advice and attention during this thesis project and for providing feedback on this thesis project report. In addition, I would like to thank my co-supervisor, Miran Hasanagić, for development support and valuable feedback on the content and structure of this thesis project report. Furthermore, I would like to thank Kenneth Lausdahl for providing assistance with the implementations and contributing to discussions along with Victor Bandur. I would also like to thank Peter Jørgensen for providing sparring on parts of this thesis project and Cláudio Gomes for discussions on Co-Simulation. Finally, I would like to thank Sigrid Mathiasen for providing support, valuable feedback on the structure of the thesis, and corrections.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis project is part of the Integrated Tool Chain for Model-Based Design of CPSs (INTO-CPS) project. One of the tools in the tool chain performs Co-Simulation, and the aim of this thesis was to investigate, whether taking advantage of concurrency could improve the performance of such a tool.

This chapter introduces the subject of this thesis project and serves as a general introduction. As this chapter serves as a general introduction, it is related to all other chapters and should be read first.

1.1. Overview

Software is increasingly being used for multiple purposes today, and one of these purposes is controlling physical processes. Such *systems* consisting of software and hardware are becoming a vital part of society, where they constitute cars, trains, medical devices and so forth. This has given rise to Cyber-Physical Systems (CPSs), which can be defined as: "*Cyber-Physical Systems are integrations of computation with physical processes*" [Lee, 2008]. It is often a multidisciplinary effort to develop a CPS, for example a combination of mechanical and software engineering. They resemble embedded systems, but are considered to be more open to the outside environment. CPSs are typically *reactive systems*, meaning they perform computations based on events and therefore keeps running unlike *transformation systems*, that performs a computation and stops.

When developing CPSs it can be useful to create *models* of components, that are part of the system. A model is an abstract description of a component, where the properties relevant to the goal is kept and others are abstracted away. Models can only approximate real components, and *fidelity* is a measure of, how close a model is to the real component. Models of hardware and software components can used in *simulations*, which may be part of developing a CPS. A type of simulation is *Co-Simulation*, which is simulation of technical systems or models created with different tools and coupled together in a semantically sound fashion. A Co-Simulation typically consists of exchanging data between the models that are part of the simulation, making them progress an amount of time, called the step size, exchange data again and so forth. As mentioned, CPSs are generally reactive systems and therefore it is necessary to set an end time of the simulation. Once the end time is reached, it is an indication of the end of a simulation and the results are reported.

Co-Simulations can use variable step size, where the steps are decided by the components, or fixed step size, where the same step size is used. If fixed step size is used, it can have a significant impact in achieving ”correct“ results of a simulation, as the real world is continuous but the simulations use discrete time.

This thesis project concerns the further development of an already existing Co-Simulation Orchestration Engine (COE), which is an application that performs Co-Simulation. In this case, the Co-Simulation is performed by utilizing the Functional Mock-up Interface (FMI), which is a tool independent standard for Co-Simulation. It defines an interface, which components, called Functional Mock-up Units (FMUs), must implement. If Co-Simulation is used in the development of a CPS, the tool performing the Co-Simulation should be fast, and this thesis focuses on determining whether implementing the COE using concurrency[1] gives any performance optimizations. It is also expected, that the COE exhibits determinism, meaning that given the same input it will always produce the same output, in order to reuse the simulations for test regression. The development effort in this thesis project consisted of two parts. The first part was to refactor the existing implementation of the COE, and the second part was to implement concurrent alternatives to the COE. The refactoring was performed to make it more suitable for running simulations concurrently and therefore prepared the COE for implementing concurrency usage. Experimentation was carried out with three different concurrent implementations: One using parallel collections, one using futures, and one using actors.

First, section 1.2, contains the author's motivation for choosing the topic of this thesis project. Afterwards, section 1.3 describes the hypothesis, that will be tested in this thesis along with other goals. Following the description of the hypothesis and goals, section 1.4 describes the scope of the thesis project. Next, section 1.5 describes the approach to testing the hypothesis and achieving the goals. Finally, section 1.6 and 1.7 presents a reading guide and the structure of this thesis report.

1.2. Motivation

The primary motivation for this thesis project was the increasing number of CPSs [Lee, 2010]. CPSs are becoming a vital part in many aspects of society and industry such as health care, transportation, and agriculture; therefore it is critical, that they behave as expected. Besides behaving as expected, it is of interest to minimize the development time of CPSs. A way to achieve both is by creating better tools for the development of CPSs, and this thesis targets the development of a tool called the COE. The COE should enable better simulation of multi-disciplinary systems compared to the existing options, that often are tailored to specific systems, and therefore cannot be applied in general [Bastian et al., 2011]. Another interesting part of the simulations is the possibility of gradually introducing real components as part of the simulation alongside models. This is referred to as hardware-in-the-loop if the components are hardware or software-in-the-loop if the components are software.

In order to minimize the development time of a CPSs using simulations it is desired to make the COE as fast as possible. As hardware makers have turned to multicore processors in order to properly utilize the new hardware possibilities [Geer, 2005; Creeger, 2005], software should take advantage of the opportunities in using concurrency. However, concurrency introduces additional complexity and challenges in the development of an application. Some of these challenges, that

[1] Concurrency and parallelism will be used interchangeably, as this thesis is not concerned with interleaving as opposed to real parallelism.

are inherent in the development of concurrent systems, can be addressed by using functional programming. The implementation of the COE was performed in Scala, which is a programming language supporting the functional paradigm, and therefore provides a solid foundation for developing with concurrency.

The author of this thesis believes, that tools must continue to improve in order for systems to improve. Development of tools also require a thorough understanding of the subject they target, and the possibility of combining tool development with CPSs is motivating. Furthermore, the usage of concurrency will only increase and the possibility of doing research on this topic while developing in a language supporting the functional paradigm is most interesting.

1.3. Hypothesis and Goals

Hypothesis: If two or more FMUs are used in a Co-Simulation performed by the COE, then the performance of executing the Co-Simulation can be improved by taking advantage of concurrency in the implementation of the COE.

The performance in this context is considered how fast a Co-Simulation is performed by the COE application, and it is therefore measured in terms of time. The reason for the constraint of two or more FMUs is, that splitting the simulation of a single FMU would not lead to performance improvements, as the functions contained in an FMU must be invoked sequentially (this will be described in more detail in chapter 2).

Besides investigating the hypothesis, other goals were set for this thesis project. These goals are related to the subject of this thesis, but also to other areas such as programming languages and personal skills. The goals were the following:

Goal 1: Understand the concepts of CPSs and Co-Simulation. Additionally, learn how Co-Simulation can be performed using the FMI standard.

Goal 2: Learn the basics of the challenges that implementing the usage of concurrency presents, and how the functional paradigm addresses these.

Goal 3: Learn and use the programming language Scala for the development of the COE application with different concurrency features. A part of this goal is to be able to generalize on implementing functionality that uses concurrency in an application.

Goal 4: Improve personal skills in general for conducting research by identifying transferable skills that may be usable in the future.

1.4. Scope

The scope of this thesis project was to understand the fundamental concept of a CPS and how Co-Simulation is performed and contribute to the development of such a system. Additionally, it was necessary to understand the FMI standard to the extent, that it could be utilized in the implementation of a COE. The COE is being implemented in the programming language Scala, and it was within the scope of this thesis project to use this language for further development of

the COE. It was also within the scope of the thesis project to understand the basics of concurrency challenges, and how concurrency can be used in Scala and in the implementation of the COE.

An application can be evaluated based on many different quality attributes such as fault-tolerance, safety, and maintainability. The implementation of the COE focuses on determinism, correctness, and performance in terms of execution time. Furthermore, the concurrency features will be evaluated based on composability, simplicity, configurability, scalability, documentation, and performance. The rating of these quality attributes are described in chapter 4.

The COE application was lacking features at the start of this thesis project, and it was out of scope to implement additional features. Therefore, it was considered out of scope to implement the possibility of performing simulations, where the dependencies between models are cyclic, where the stability of simulators are validated, and where the simulators perform the so-called zeno effect. Neither were extrapolation or interpolation of results in case of different step sizes to be supported. This thesis project only used models for simulations, and it was also not part of the scope to compare results of simulations using models to results using the realized systems. The use of concurrency features was also constrained to the default threading strategies. Furthermore, as the COE was written in Scala at the start of this thesis project, the programming in this thesis was also performed in Scala.

The reader of this thesis is expected to have a background in information and communications technology in order to understand the concepts, that are used. Additionally, it is expected that the reader has basic knowledge of functional programming and thereby knowledge of common functions such as map and filter.

1.5. Approach

The approach to investigating the hypothesis defined in the previous section is presented in figure 1.1 and was used to confirm or refute the hypothesis. Furthermore, the approach helped to fulfill the goals set for this thesis. The steps are explained in detail below.

Litterature Study: To understand the subjects CPSs, Co-Simulation, and FMI it was necessary to read relevant literature. To gain a thorough understanding of concurrency, it was necessary to understand the challenges inherent in implementing concurrency usage by reading literature on this as well.

Baseline Establishing: The implementation of the COE at the start of this thesis project was an unstable prototype, and it was therefore necessary to fix various bugs as the first step in order to make it stable. Once this was done the baseline in figure 1.1 was established.

Concurrency Implementations: Once a baseline had been established, a new implementation was made, that was separated from the baseline as shown in figure 1.1. This implementation was a refactored version of the baseline, and the purpose of the refactoring was to prepare the implementation for concurrency. Afterwards, functionality that uses concurrency was added to the refactored baseline. This resulted in three different concurrency implementations: One using parallel collections, one using futures and one using actors.

Comparison and Evaluation: It was necessary to compare the effect of the concurrency implementations versus the baseline, to see whether any performance improvements were present.

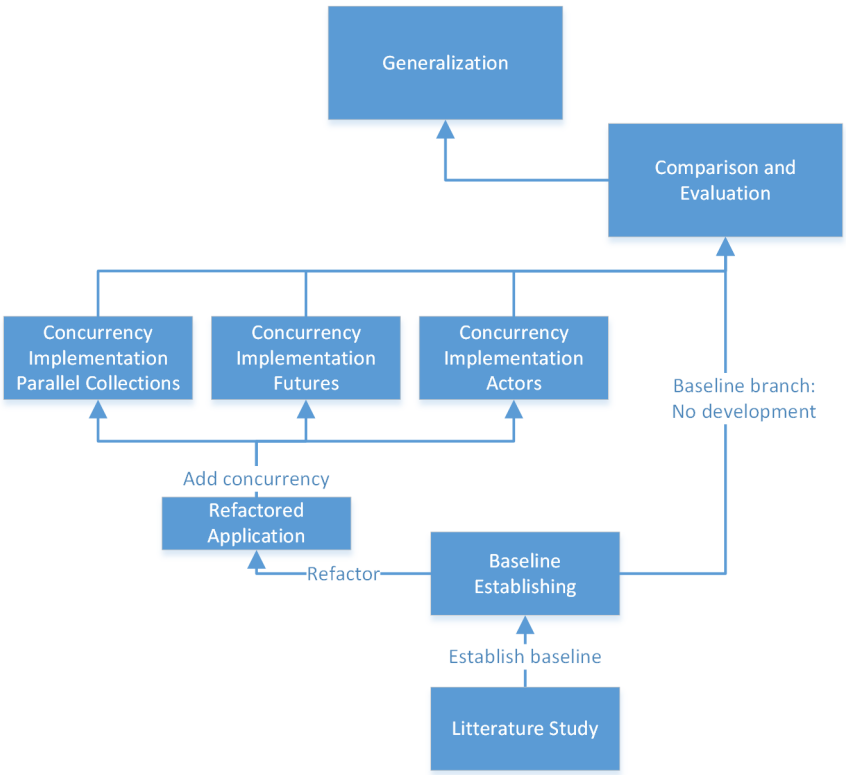


Figure 1.1: The approach to investigating the hypothesis and fulfilling the goals.

Therefore several tests were implemented to understand and show different cases where the performance was increased, decreased or stayed the same with concurrency as opposed to without. The tests were also used to verify, that the concurrency implementations were semantically equivalent and thus provided the same results as the baseline.

To decide whether concurrency is the right direction for the COE it was necessary to evaluate and discuss the results. This helped to outline the future work of the COE.

Generalization: To use the analysis carried out in this thesis project in a broader context, the process of applying concurrency to an existing application was generalized to a set of guidelines.

1.6. Reading Guide

References
References are referred to by author and year, which corresponds to an entry in the bibliography in this thesis. As an example, [Lee, 2008] refers to a reference entry in the bibliography with the same tag.

Emphasis
Emphasized words are written in *italic*, such as *emphasized*.

Quotations
A quotation from literature is written in *italic* and placed in double quotation marks, such as "*This*

is a quote".

Numbering of figures, listings, and tables
Figures, listings, and tables are presented using using the convention figure/listing/table <C>.<N>, in which C refers to the chapter number, and N is the order of the figure/listing/table in chapter C.

Code elements
Elements that are part of code, e.g. Scala or Java, are written in a `typewriter font`. This is also used to separate a concept from a specific implementation of the concept in a programming language, as the concept will be written regularly as: actor, whereas when referring to the specific type that is used in a programming language it will be written as `Actor`.

Listings
Different listings are used throughout this thesis and they have different layouts. The layout in the appendices have the programming language clearly stated in the caption as shown in listing 1.1. The layouts in the report will have the programming language stated in the top left. Listing 1.2 shows the Java layout, listing 1.3 the Scala layout, listing 1.4 the XML layout, listing 1.5 the JSON layout, and listing 1.6 shows the pseudocode layout. Two threads might be illustrated in one listing to emphasize interleaving and order of events. This will be expressed as shown in listing 1.7. Thread 1 performs an event first, then Thread 2 performs an event, and then both threads perform an event without order. However both events on line 3 would be performed before any event on line 4+. In this example and in general, if there are no line numbers on a given line, then it is only inserted for explanatory purposes and not part of the code. Line wrapping is an exception to this rule and it is illustrated with a red arrow as shown in listing 1.3.

```
1 // A Java comment
2 public static void main(String args[]) {
3 System.out.println("Hello, World!"); }
```

Listing 1.1: Java, Example of Java block in appendices

Java

```
1 // A Java comment
2 public static void main(String args[]) {
3   System.out.println("Hello, World!"); }
```

Listing 1.2: Example of a Java code block

Scala

```
1 // A Scala comment
2 def main(args: Array[String]) {
3   println("Hello, World! A long line to illustrate line
      ↪ wrapping.")   }
```

Listing 1.3: Example of a Scala code block

XML

```
1 <!-- This is an XML comment -->
2 <message type="test"> Hello, World! </message>
```

Listing 1.4: Example of an XML block

JSON

```
1 { "message": "Hello, World!" }
```

Listing 1.5: Example of a JSON block

Pseudocode

```
1 print "Hello, World!"
```

Listing 1.6: Example of a Pseudocode block

Pseudocode

```
  //Thread 1                                  //Thread 2
1 Print Thread 1
2                                             Print Thread 2
3 Set x to 2                                  Set x to 5
```

Listing 1.7: Example of a threaded Pseudocode block

1.7. Structure

The structure of this thesis is presented in figure 1.2. Every chapter is a solid box, and the arrows between them indicate their dependencies. As it can be seen in the figure, the background chapter is relevant to most of the chapters in this thesis. The text in the left hand side of the figure indicate the main theme of the chapters in the swim lane, which is marked by horizontal lines.

Chapter 2: This chapter presents the background of the concepts leading up to this thesis and the concepts underlying the implementation. Because it presents background on these different concepts, it is relevant to chapters 3 to 5.

Chapter 3: This chapter describes the baseline implementation, the refactoring of the baseline, and the implementations using concurrency. Thereby, chapter 3 establishes the basis of the implementations that are evaluated in chapter 4.

Chapter 4: This chapter tests and evaluates the implementations in chapter 3 and provides a foundation for the guidelines in chapter 5.

Chapter 5: This chapter contains generalized guidelines on applying concurrency to an existing application. This can be useful in the development of other applications that seek to use concurrency, as it contains steps and advice on this topic learned during the work that was carried out in this thesis project. The chapter contains references to other chapters, but is roughly self-contained.

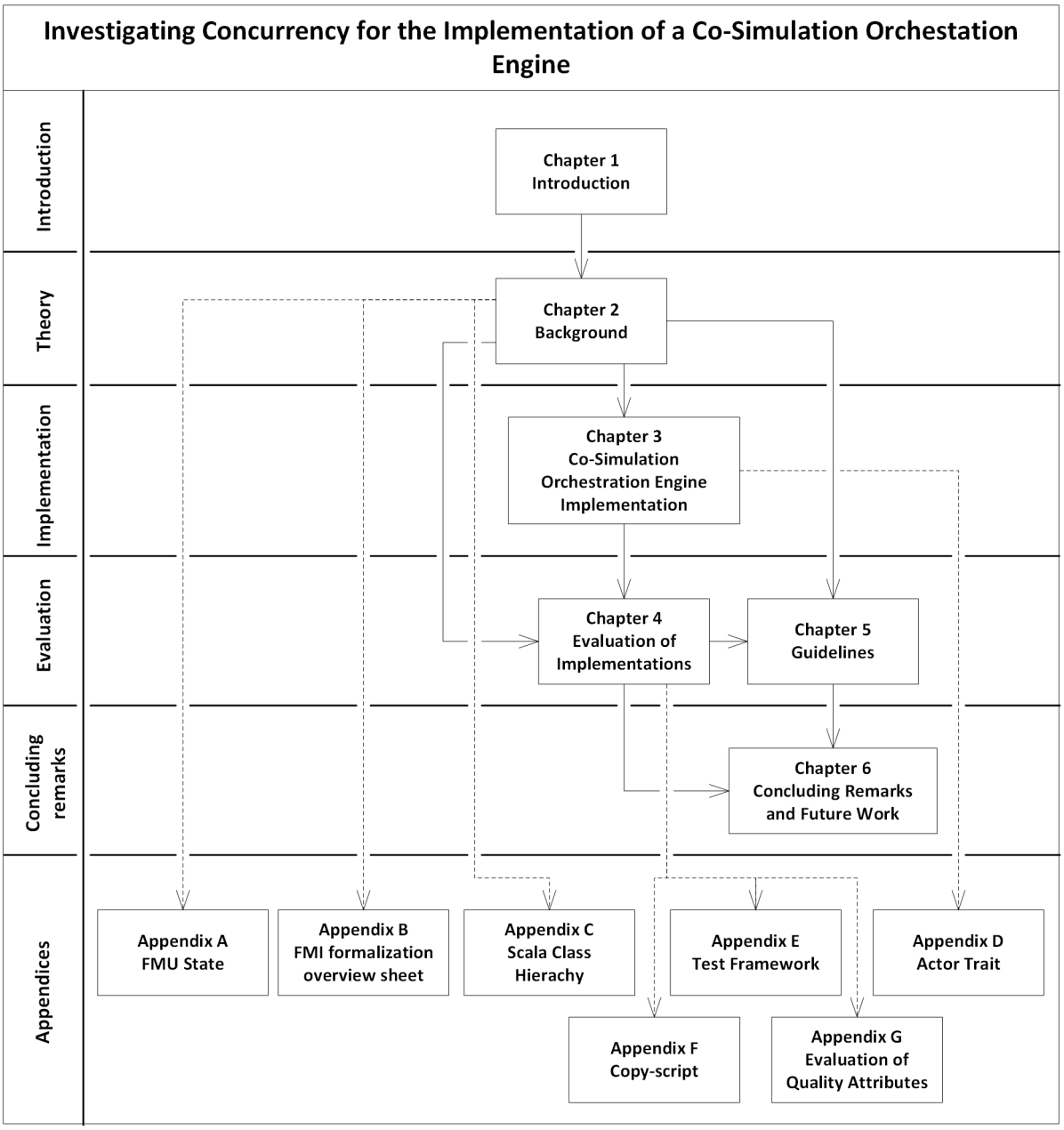


Figure 1.2: The structure of this thesis.

Chapter 6: The chapter concludes the work conducted in this thesis project along with evaluating the hypothesis and goals presented in section 1.3. Furthermore, ideas on future work are discussed.

Appendix A: This appendix shows the state machine used for Co-simulation with FMI and is used in the presentation of the FMI interface in section 2.4.

Appendix B: This appendix contains an overview of the formalization presented in an article on algorithms for FMI used in section 2.5.

Appendix C: This appendix contains the Scala class hierarchy and is used to introduce Scala in section 2.8.

Appendix D: This appendix contains the implementation of an `Actor` trait, that is used in the concurrent implementation using actors in section 3.3.4.

Appendix E: This appendix contains the implementation of the test framework, that is described in section 4.3.

Appendix F: This appendix contains a shell script used for generating test scenarios. This is used in chapter 4.

Appendix G: This appendix contains the evaluation of the quality attributes, that is described in section 4.7.

The Master's thesis report is also attached within the CD, if the reader wishes to have it as a pdf file. Furthermore, the CD contains the source code and FMUs used in this thesis project along with the test results.

Chapter 2

Background

This chapter introduces the concepts Cyber-Physical Systems (CPSs) and Co-Simulation using the Functional Mock-up Interface (FMI) that this thesis builds upon. These concepts were investigated in a literature study, that focused on understanding the respective concepts. This knowledge is useful in order to understand the tasks, that the tool performing Co-Simulation must accomplish and how the FMI can be used for this. Additionally, literature on developing software that uses concurrency was studied and is presented in this chapter. This also contains challenges inherent in developing software that uses concurrency and how these challenges can be addressed.

The knowledge achieved within these concepts forms the basis of the existing implementation of the Co-Simulation Orchestration Engine (COE) and the implementation using concurrency described in chapter 3 and evaluated in chapter 4. Furthermore, this knowledge was used to create guidelines on introducing the usage of concurrency in an existing application in chapter 5. Lastly, this chapter helps to fulfill the following goals defined in chapter 1: Goal 1 on understanding CPSs and Co-Simulation using FMI; goal 2 on learning the basics of challenges posed by concurrency and finally, goal 3 on learning the programming language Scala.

2.1. Introduction

Cyber-Physical Systems (CPSs) combines computation and physical processes and are used in many areas today such as automotive, aerospace, and energy. These systems are often networked and consists of multiple components, and therefore timing presents a challenge along with dependability, which covers several attributes such as reliability, availability etc. These challenges can lead to nondeterminism, which makes the systems difficult to reason about and use.

A way of addressing these challenges is simulating the components that make up the system or models of these along with the interactions between them. This is called Co-Simulation, as several coupled systems are simulated. This can help to detect instability in the system or discover undesired behavior. However, different simulation tools perform different simulation tasks [Bastian et al., 2011], and this leads to system specific solutions, that are difficult to reuse in the simulation of other systems. To make simulations more applicable in general, a standard called the Functional Mock-up Interface (FMI) was created [FMI development group, 2014a]. This standardizes how components must be packaged and interfaces they must implement in order to use them for

Co-Simulation using FMI without requiring specific tools. When performing Co-Simulation, determinism is important, so the simulations are reusable. Therefore, several algorithms specific to the FMI standard have been designed for orchestrating Co-Simulations to secure deterministic behavior of the Co-Simulation Orchestration Engine (COE) responsible for executing the Co-Simulations.

As simulating is expected to be a reoccurring task in the development of CPSs, it is desirable that these are executed fast. Because processors today have multiple cores [Geer, 2005; Creeger, 2005] it may be possible to increase the performance of executing simulations by using concurrency. However, concurrency is inherently nondeterministic and therefore presents additional challenges in achieving determinism. To solve or simplify these challenges, Scala adheres to the functional paradigm and provides concurrency features that can be taken advantage of.

The following section introduces CPSs, and Co-Simulation using FMI is described in section 2.3 and 2.4. Next, section 2.5 describes algorithms on how to perform such a simulation deterministically. As this thesis project emanates from the project INTO-CPS, it is introduced in section 2.6. To be able to take advantage of concurrency, it is necessary to understand the challenges it presents, and some of these are presented in section 2.7. Lastly, section 2.8 introduces the Scala programming language and different concurrency features available in Scala.

2.2. Cyber-Physical Systems

This is a broad subject, as many systems today are CPSs e.g. cars, trains, robots, and medical devices. It is not a new concept as such, and many will think of it as embedded systems. However, CPSs is a broader term and it also includes the Internet of Things. Whereas embedded systems are considered closed for the outside environment, CPSs can be networked and closely linked to physical elements. In the article "CPS Foundations" Lee states: "... *Cyber-Physical Systems arguably have the potential to dwarf the 20th century IT revolution.*" [Lee, 2010]. The use of CPSs will also grow as computers become cheaper and integrated into more products. One of the challenges in CPSs is the cross-disciplinary nature and this requires engineers from different fields to come together and design the systems.

A significant challenge for CPSs is *dependability*, which can be defined as: *"... the ability to deliver service that can justifiably be trusted"* [Avizienis et al., 2004]. Trusted can then be defined as: *"the ability to avoid service failures that are more frequent and more severe than is acceptable"* [Avizienis et al., 2004]. Dependability covers several attributes of a system such as availability, reliability, safety etc. Lee adds predictability to the challenges for CPSs and links it closely to reliability, which is one of the attributes covered by dependability. This is because systems are based on predictions in a controlled environment, and therefore can be made reliable. For example, a smart TV exists in a controlled environment, and can therefore be made reliable. However, the physical world is in general not predictable, and therefore reliability becomes much more difficult as unexpected conditions can happen, and some systems such as planes or medical devices may not be allowed to behave undesirably. Lee sets up a simple principle [Lee, 2008]:

> *Components at any level of abstraction should be made predictable and reliable if this is technologically feasible. If it is not technologically feasible, then the next level of abstraction above these components must compensate with robustness.*

An example is, that it is difficult to create reliable and predictable wireless links and therefore several protocols exists to compensate accordingly. As no component is perfectly dependable, combining components in CPSs can lead to undesired results, and it can therefore be necessary to define an appropriate balance.

Lee also mentions abstractions with relation to timing as a significant issue. Our computers become faster, but that is not necessarily what is needed for CPSs, instead it is essential that physical actions takes place at the right time [Lee, 2010]. As software abstractions are made, complex details are chosen to be abstracted away. An example of this is C, the popular programming language for embedded development. It does not contain any notion of timing semantics, meaning it can execute in the right fashion but perform wrongly. This means the logic is correct, but the operations are carried out at the wrong points in time and therefore it does not perform as desired. As more abstractions are made and the complexity of the applications grows, e.g. by adding inter-process communication, it becomes increasingly difficult to control timing. In general, most programming languages do not contain timing requirements, and therefore the abstractions have failed, because they lack features to solve these challenges. This is acceptable in many cases, but in critical CPSs it is not. Therefore Real-Time Operation Systems (RTOS) deal with worst case execution time, which introduces a significant overhead.

Lee argues, that part of the solution is better concurrency handling than using threads. Concurrency cannot be avoided because many things in the physical world are interleaved, yet threads are a bad abstraction as they are difficult to comprehend for people even though the physical world is well-understood. By using other programming languages that support concurrency better some of these challenges could be overcome. A bottom-up approach is suggested, where timing is thought of from the beginning. However, this could mean re-thinking progress in many years of software evolution because timing has been left out. A top-down solution is generating software from models, that contain the required system behaviors. The CPS assurance framework in figure 2.1 uses a model in the ”Conceptualization Facet“. This is used to capture the behavior of the system, and therefore it is important, that this model approximates the real system to a degree, where it becomes usable. This is called *model fidelity*, and it is a measure of the realism of a model [SISO, 1999]. The ”Realization Facet“ is the realization of the model, and finally the ”Assurance Facet“ is where it is proven, that the realized system actually works as it should, and how the model stated it should. However, an adage goes: ”*All models are wrong, some are useful*“ [Box and Draper, 1987; Newcombe et al., 2014].

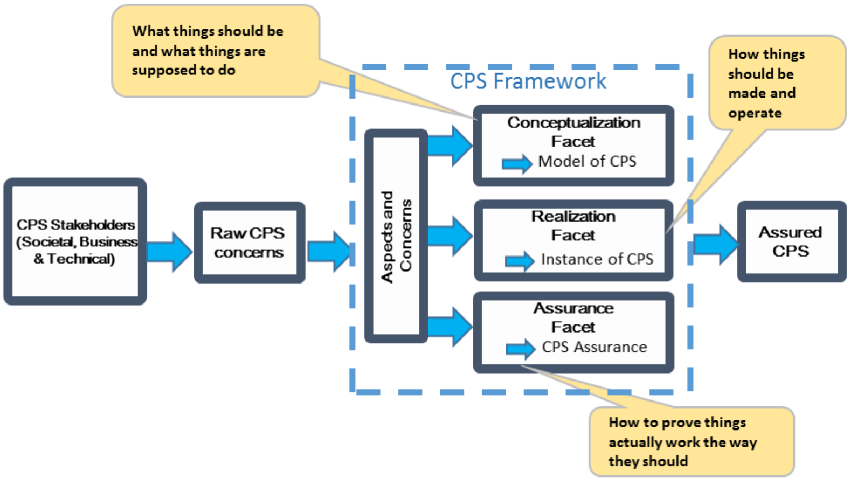


Figure 2.1: An assurance framework for CPSs [NIST, 2015].

2.3. Co-Simulation

Co-Simulation is well suited for usage in CPSs, as Co-Simulation is simulation of coupled technical systems or constituent models created with different tools. Each constituent model handles an element of the full system, which makes it possible to make virtual products. For example, a system consisting of many virtual components that together make up a virtual car. According to Bastian et al. many complex multi-disciplinary systems cannot be modeled naturally in one simulation tool alone, but require several specialized simulation tools that each do their part. This is because some components are most naturally described using differential equations whereas others are more naturally modeled as discrete event or discrete time systems [Bastian et al., 2011]. This means it has been necessary to create solutions tailored for a specific purpose instead of having a general solution, and this can be expensive.

It is possible to use a *master-slave* relationship for Co-Simulation, where the slaves are responsible for simulating the individual components and the master is responsible for orchestrating the simulation and transferring data between the slaves. Orchestrating the simulation means to simulate the components in the correct order, map outputs from one slave as input to another slave correctly, and transfer the data at restricted discrete communication points. In order to engage in such a Co-Simulation, a slave/simulator S must be able to exchange data during the simulation at specific communication points in time, as shown in figure 2.2, where u is input and y is output.

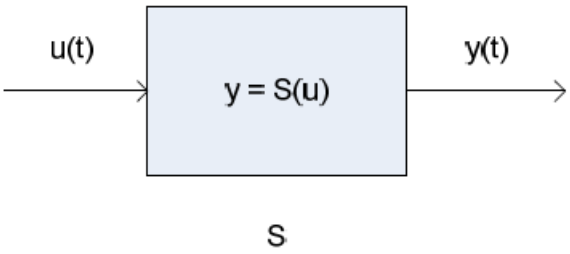


Figure 2.2: A block representation of a simulator [Bastian et al., 2011].

Simulators/slaves can have different capabilities such as variable step sizes[1], rollback mechanisms[2], asynchronous/synchronous, discrete events or/and differential equations etc. This puts different requirements on the simulation algorithm used by the master, called the Master Algorithm (MA), which van Acker et al. presents a solution to [van Acker et al., 2015]. The idea is to generate an optimized orchestration algorithm based on analysis of the model that is to be simulated. For example if slave2 depends on the output from slave1 but they have different step size, then the optimized algorithm will interpolate or extrapolate the simulation values from slave1 before passing them to slave2. The overall flow of a simulation usually consists of three phases:

Initialization: The master gets the properties of the slaves, chooses an algorithm, prepares the slaves and establishes the communication channels. As part of this state it is also necessary to either pass a port dependency graph to the master or letting the master construct its own graph based on configuration files. Figure 2.3 presents an example of connections between the slaves via their input/output ports.

[1] A step size is an amount of time, and when a simulator does a step it goes an amount of time forward.

[2] A rollback is when a simulator rolls back to a previous defined state, meaning all relevant information is reverted to its value at that state.

Simulation phase: The master sets input values on the slaves and invokes them to run a simulation with a specific time step. The slaves must respond with a status whether the step was accepted or not. In this phase, it can be necessary to perform a rollback (if possible) for the relevant simulator and run the simulation again with a different step size. The master gets the output values from the slaves after a simulation.

Tear down: The master ends the simulation and shuts down the simulators, releases memory, and reports the results along with potentially performing other tear down tasks.

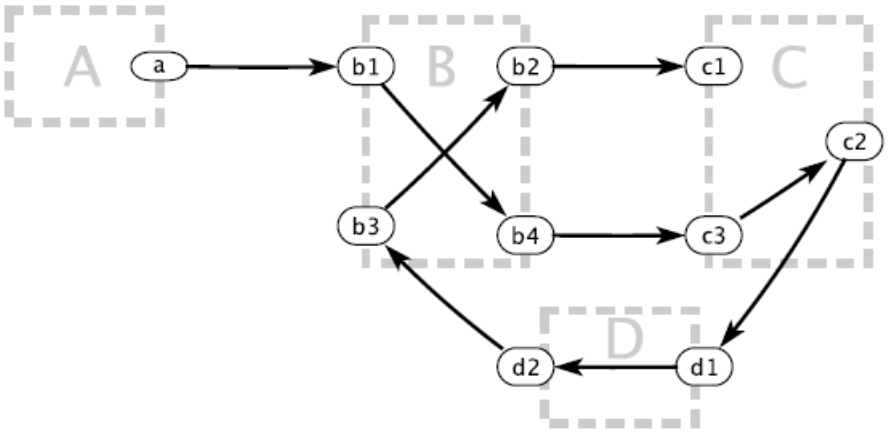


Figure 2.3: Example of a simulated CPS with dependencies between simulators (the gray boxes) via their respective ports (the black ellipses) [Broman et al., 2013].

2.4. The Functional Mock-up Interface

As mentioned above, it is a challenge to combine different simulation tools and keep the effort of developing custom simulations down. The FMI was created to solve these challenges, as it is a tool independent standard for the exchange of models and Co-Simulation. It is the result of the ITEA2 European project called MODELISAR with the purpose of improving the design of systems and embedded software in vehicles [ITEA Office Association, 2015]. The project was started in July 2008 and finished in December 2011 and the core development partners are now part of the ”Functional Mock-up Interface“ project under the Modelica Association. This project has the purpose of developing, standardizing and promoting the FMI in the context of using models of dynamic systems with Software/Model/Hardware-in-the-Loop simulations for CPSs in general. FMI 1.0 was released in 2010 and is supported by 66 tools; FMI 2.0 was released in July 2014 and is supported by 19 tools [FMI development group, 2014b]. Version 1.0 was split into a Model Exchange part and a Co-Simulation part. Version 2.0 merged these into one standard with two parts, which enables sharing of functions relevant to both. FMI for Model Exchange and for Co-Simulation is described below, and afterwards the remainder of this thesis will focus on FMI for Co-Simulation.

FMI for Model Exchange: A modelling environment can generate C code of a dynamic system in the shape of an input/output block that can be used by other modelling and simulation tools. These models do not necessarily contain their own solvers.

FMI for Co-Simulation: The concept here is to couple two or more models that each have their own solvers and data exchange is restricted to discrete communication points. An MA controls the data exchange and synchronization between individual models.

The standard provides and describes C interfaces, that can be implemented based on requirements; some functions are necessary to implement and others are optional. A component implementing FMI is called a Functional Mock-up Unit (FMU), and an example can be seen in figure 2.4. An FMU must also provide an FMU configuration file, which is defined in the FMI standard. Some existing solutions have implemented FMUs with success:

- A Modelica-based FMU has been integrated into three other modeling and simulation environments tools: GridLAB-D, TRNSYS and High Level Architecture (HLA) with promising results [Elsheikh et al., 2013].
- The integration of an in-house tool with FMU exports from a Modelica model reached a clear improvement of the productivity. However challenges still exists, e.g. the FMU acts like a black box and it is difficult to identify errors in the model [Sun et al., 2011].

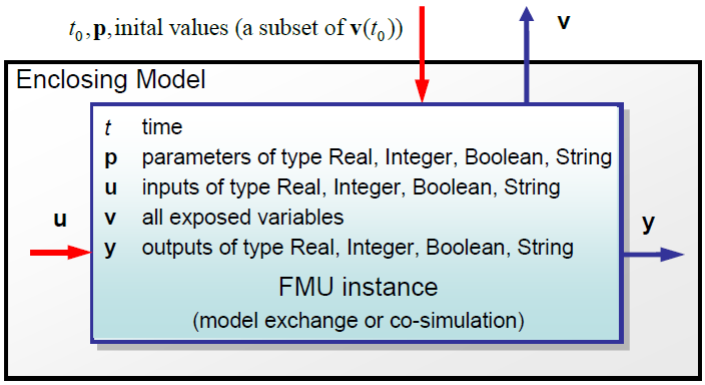


Figure 2.4: Schematic overview of an FMU. The blue arrows are information provided by the FMU and the red arrows are information provided to the FMU [FMI development group, 2014a].

2.4.1 Functional Mock-up Unit Package

An FMU is packaged in a zip-file with the extension `.fmu`, which contains all necessary information required to utilize the FMU. The files included in such a package are:

An FMU configuration file: The FMU configuration file contains a unique identifier for the FMU, variable definitions such as inputs, outputs, start values and so forth. A schematic overview can be seen in figure 2.4 and an example of the variables in a FMU configuration file can be seen in listing 2.1. The FMU containing the configuration file shown in listing 2.1 is a model of a water tank that fills up with water when the valve is closed and drains when it is open. It contains two variables: an output variable describing the water level, see line five to nine, with an initial value of one, see line eight, and an input variable with the state of the valve, see line 10-14, with an initial value of zero, see line 13.

FMU implementation: The realized FMU implementation is either a library or the source files necessary to build the FMU. It is possible to package libraries for different platforms.

Miscellaneous: This can be documentation, additional libraries or anything considered important for the usage of the FMU.

XML

```
<fmiModelDescription fmiVersion="2.0" modelName="tank"
guid="{8c4e810f-3df3-4a00-8276-176fa3c9f001}"
numberOfEventIndicators="0">
<ModelVariables>
<ScalarVariable name="level" valueReference="0"
description="the tank level" causality="output"
variability="continuous" initial="approx">
<Real start="1" />
</ScalarVariable>
<ScalarVariable name="valve" valueReference="1"
description="the tank valve state" causality="input"
variability="discrete" >
<Boolean start="0" />
</ScalarVariable>
</ModelVariables>
</fmiModelDescription>
```

Listing 2.1: Example of variables in an FMU configuration file.

2.4.2 Functional Mock-up Unit Usage

There are certain steps involved in using an FMU, which are shown in a simplified version in figure 2.5; a more detailed state machine and table describing allowed state changes and function calls can be seen in appendix A. The `doStep` function is invoked on the FMU with a $step\ size\ (milliseconds) > 0.0$ and it is a request to the FMU to go from $current\ time$ to $current\ time + step\ size$, which becomes the new $current\ time$ once the step is completed. The simulation is finished once the configured end time for the simulation is reached: $current\ time + step\ size > configured\ end\ time$. In version 2 of the FMI standard it was made possible and optional whether to include functions to get and set state. As it is possible to request how much of a step an FMU was able to complete, $t_{complete}$, the possibility of setting and getting states makes it possible to perform a roll back to the previous state and use $t_{complete}$ as the new step size. It can also be useful for storing the final state to be used later on to begin a new simulation, which is mentioned as a challenge in a case study using version 1 of the FMI standard [Sun et al., 2011].

2.5. Functional Mock-up Interface Algorithms

The previous section described a single FMU and how to use it, but to use FMI for Co-Simulation in a broader context it is necessary with a master (see section 2.3) capable of handling multiple FMUs. As this thesis focuses on FMI for Co-Simulation, the term *Co-Simulation Orchestration*

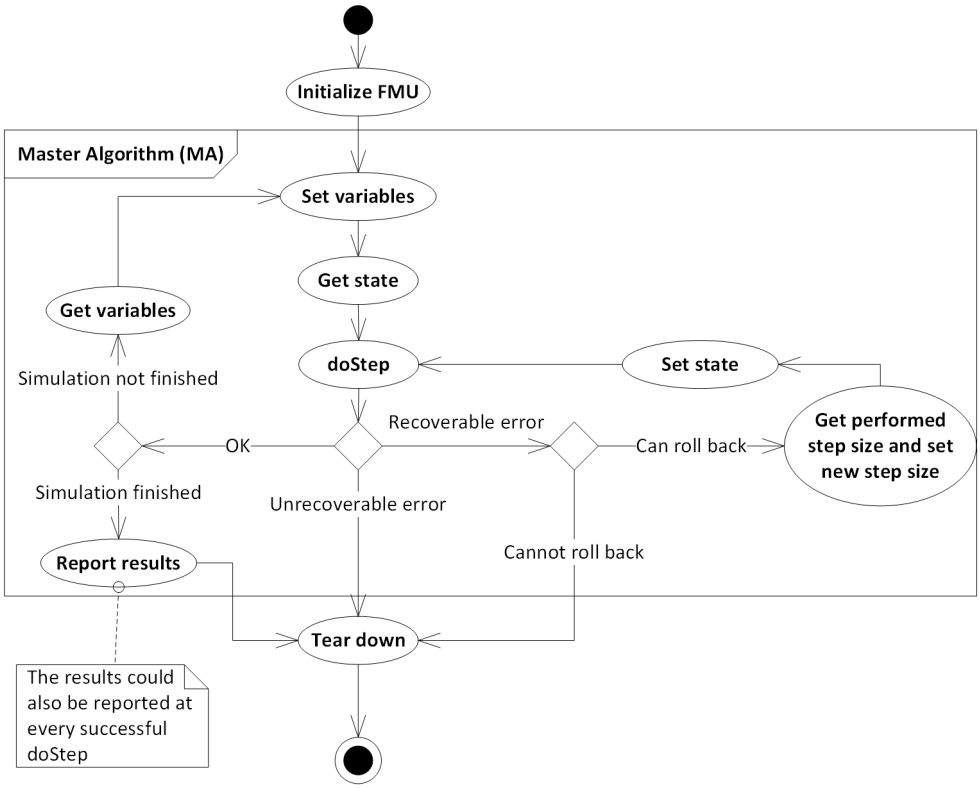


Figure 2.5: Simple Co-Simulation flow chart for a FMU.

Engine (COE) will be used instead of master below. COE is also the name of the application, which this thesis project extended upon, and it is implemented using the algorithms described in this section. The diagram in figure 2.6 presents an overview of a COE.

To achieve deterministic execution of FMUs Broman et al. describes algorithms for processing FMU configuration files, COE configuration files and for the design of MAs [Broman et al., 2013]. A COE configuration file contains the configuration of a COE and information on which FMUs to use and how they are connected, like in figure 2.3. This section will present these algorithms, and an overview sheet to help understand the algorithms can be seen in appendix B. The MAs presented below controls the flow in the ”Master Algorithm“ frame in figure 2.5 and requires the FMUs to satisfy the following constraint: If an FMU can complete a step with time size h, then it can also complete a time step with size h', where $0.0 < h' \leq h$. If the FMUs do not comply with this constraint it will be difficult to find a step size that all FMUs can perform, because it will end up as trial-and-error attempts to find a step size accepted by all FMUs.

2.5.1 Preprocessing Algorithm: Order-Variables

The Order-Variables algorithm is a preprocessing algorithm that takes place in ”Initialization“ in figure 2.6. The algorithm is described using the formalization in appendix B, and it is useful because it can take dependencies as inputs and generate an ordered list of variables to access by the MA. In smaller models it is easy to generate a list of variables in the correct order, but when models become large it is necessary to run an automated algorithm that lists the dependencies in a format that the MA can understand. The correct order describes the order to access the variables, such that the inputs are set before the outputs are retrieved.

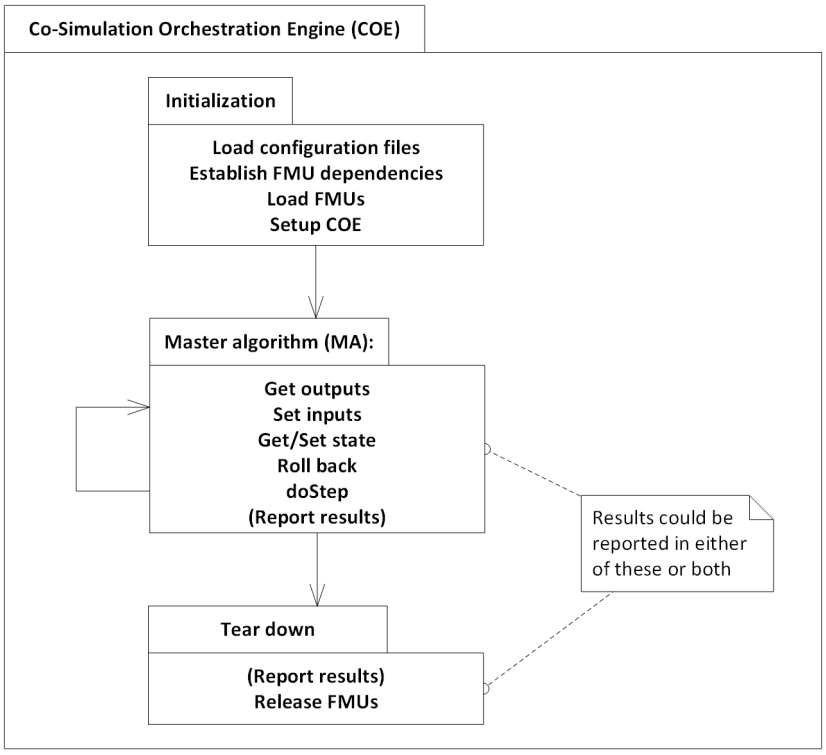


Figure 2.6: COE overview.

The algorithm requires the following inputs:

Port mapping $\mathbf{P : U \rightarrow Y}$: This describes a total function (P) between the inputs (U) and their corresponding outputs (Y) of the FMUs in a model.

Global dependency relation $\mathbf{D = \cup_{c \in C} D_c}$: This is the set of all internal dependencies of all the FMUs in the model, but not the dependencies between FMUs. C is the set of all FMU instances in a model, and D_c above is the dependencies for a particular FMU c, where $c \in C$. $\mathbf{D_c \subseteq U_c \times Y_c}$, where U_c is the set of input variables for a given FMU instance, and Y_c is the set of output variables for a given FMU c, where $c \in C$. For example to describe component B in figure 2.3 the dependency relation would be $D_{c=B} = \{(b1, b4), (b3, b2)\}$.

Global set of variables $\mathbf{\mathbb{X} = U \cup Y}$: The set of all input and output variables in the model.

The Order-Variables algorithm is based on graph theory and represents vertices by port variables $\mathbb{X}$ and an edge by $e \in \mathbb{X} \times \mathbb{X}$, which is a variable dependency. All edges E are constructed by $E = D \cup \{(y, u) | u \in U \wedge P(u) = Y\}$. This means the dependencies in each FMU unioned with the dependencies between FMUs determined by the port mapping function. The second and last step is to perform a topological sort, which will result in an error if the graph is cyclic. If the graph is cyclic the correct evaluation order cannot be derived. Bastian et al. presents an idea on how to handle cycles by creating super simulators [Bastian et al., 2011].

2.5.2 Master Algorithm: Master-Step

The Master-Step algorithm for doing a step with rollback is conceptually simple, and a flow chart is presented in figure 2.7. It is a prerequisite for this algorithm, that the dependency graph mentioned in section 2.5.1 is acyclic. The algorithm consists of two parts, the first is based on the preprocessing algorithm ”Order-Variables“ where each input is set. The second part deals with performing a `doStep` for every FMU. If all FMUs do not support the initial step size, then the FMU that performs the shortest step will eventually decide the step size, and the other FMUs will roll back and perform a `doStep` with this step size, referred to as the minimum step size. The state is stored in step 4 in figure 2.7 instead of in step 1 because otherwise step 1 to 3 would have to be run again in case of a rollback. It is possible to run this algorithm with a maximum of one FMU not supporting rollback, but it comes with the following constraint: The FMU not supporting rollback must either have the minimum step size or be equal to the minimum step size, otherwise another FMU will decide a lower step size, and the FMU not supporting rollback cannot roll back and therefore cannot continue.

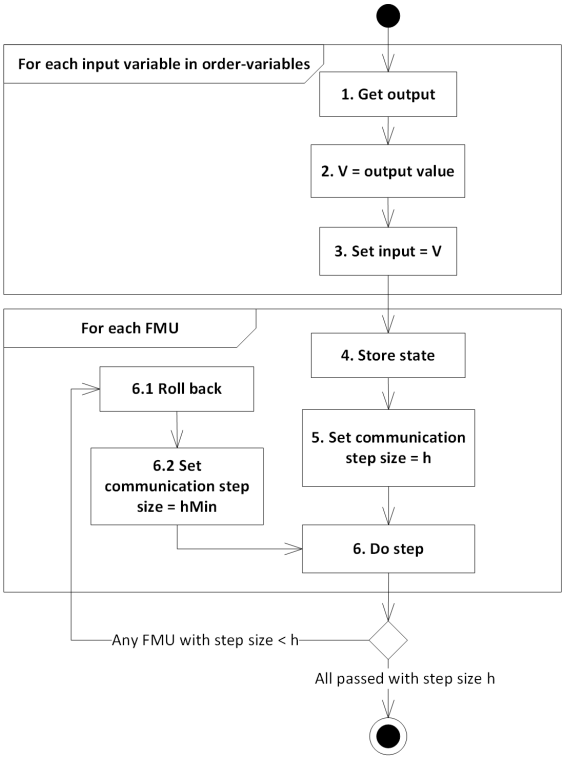


Figure 2.7: Master-Step flow chart described in section 2.5.2.

2.5.3 Master Algorithm: Predictable Step Sizes

The Predictable Step Sizes algorithm is shown in figure 2.8 and it is a prerequisite for this algorithm, that the dependency graph mentioned in section 2.5.1 is acyclic. This algorithm contains a suggested extension to the FMI standard, namely a new function called `fmi2GetMaxStepSize`. This function returns an upper bound on the step size a given FMU accepts for the next simulation step. However, it cannot be expected that all FMUs will contain this ability, so it is possible to end up with some FMUs supporting rollback (C_R), some supporting the suggested extension

`fmi2GetMaxStepSize` (C_P), and some legacy FMUs (C_L) not supporting either. If an FMU supports both rollback and the extension method then it belongs to the C_P category. The algorithm works with a maximum of one legacy FMU. The idea is first to invoke `fmi2GetMaxStepSize` on the C_P FMUs and find the minimum step size, then use this step on the C_R FMUs. If any of the FMUs belonging to category C_R cannot perform a step with this size, then roll them back and perform a step with the minimum step size, that they were able to complete. After this, perform the step on the legacy FMU. If this fails, then rollback the C_R FMUs and rerun with the legacy FMU step. Run the C_P FMUs as the last step, as these should not be able to fail. They should not fail because it is assumed that any FMU in C_P can complete any step with time less than or equal to their maximum step size. The C_R FMUs are run before the legacy FMU to prevent a failure. If they are run in the same step and one of the C_R FMUs makes a step size smaller than the legacy FMU, then it is impossible to roll back the legacy FMU, and thereby the process will have failed.

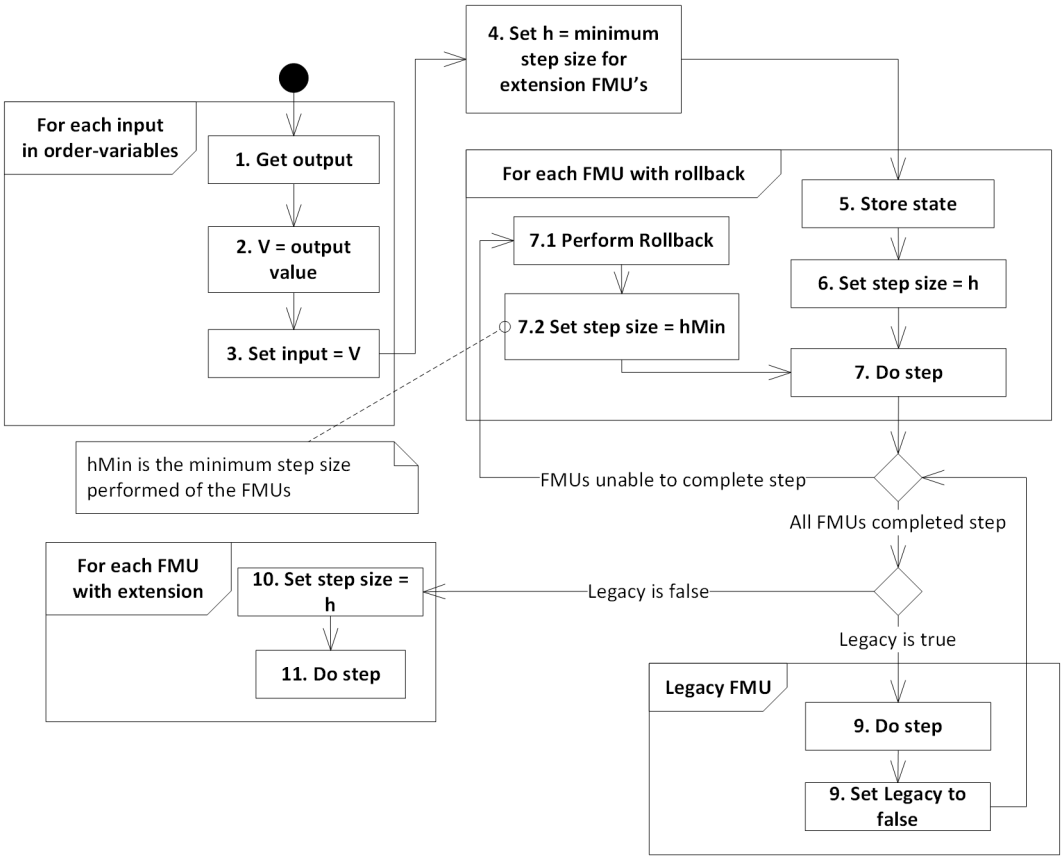


Figure 2.8: Master-Step with predictable step sizes flow chart described in section 2.5.3.

2.6. INTO-CPS

This thesis is related to the Integrated Tool Chain for Model-Based Design of CPSs (INTO-CPS) project [Larsen, 2015], which is an EU funded project to support multidisciplinary and collaborative modeling of CPSs covering the full life cycle. The consortium behind INTO-CPS consists of seven industrial and four academic partners, who provide knowledge, baseline technologies

and applications. The purpose is to create a chain of interlinked tools, based on FMI for Co-Simulation, that supports development of CPSs from requirements to realization in mechanical and electronic hardware along with software. Figure 2.9 illustrates the overall workflow and services that should be offered by the tool chain and how they are expected to be used in the development of CPSs. This thesis contributes to the tool responsible for performing ”MiL Co-Simulation“ and ”HiL / SiL Simulation“, which is simulations with models and hardware/software in the loop respectively. Requirements from the industrial case studies will drive the development of a master, that must exchange data between the models and provide interfaces to other tools. These case studies will also be used to evaluate and demonstrate the tool chain. It is also part of the project to provide a formal semantic basis for co-modeling.

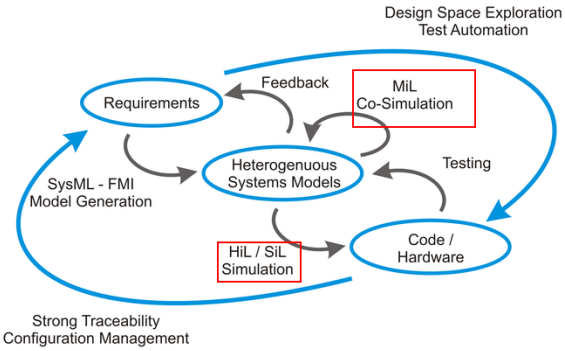


Figure 2.9: Overall workflow and services offered by the INTO-CPS tool chain. The contribution from this thesis is marked in red squares.

2.7. Challenges in Concurrency

When developing CPSs using Co-Simulation, it is desirable to execute the simulations as fast as possible. The simulations are used to verify the behavior of the components in combination, and verification can lead to the discovery of undesired behavior. The faster undesired behavior can be identified, the faster it can be resolved. As many processors today have multiple cores [Geer, 2005; Creeger, 2005], using concurrency may increase the performance of an application. However, concurrency is inherent nondeterministic and therefore synchronization mechanisms, i.e. the semaphore, have been created to ensure determinism when it is required [Dijkstra, 1965a]. This section presents a definition of concurrency and describes some challenges that applying concurrency and using synchronization mechanisms presents. These challenges are important to understand in order to successfully implement the usage of concurrency in the COE.

2.7.1 Definition

Concurrency is considered a property of a system, where the execution of a thread occurs concurrent with the execution of other threads. A thread executes sequentially, meaning that given the same input it will go through the same sequence of actions [Dijkstra, 1971], and independently of the execution of other threads, yet shares the same address space. Parallel Programming is described by Gill as:

> *By "Parallel Programming" is meant the control of two or more operations which are executed virtually simultaneously, and each of which entails following a series of instructions where operations are executed virtually simultaneously.* [Gill, 1958]

Rochester describes it more abstractly as "*The trick used is called 'multiprogramming' and it involves doing two or more different jobs at once.*" [Rochester, 1955]. Parallel Programming, Multiprogramming and concurrent programming are considered to be the same in this thesis. Thus implementing a system using Parallel Programming leads to a system with concurrency. It is becoming a vital part of software development as the increase of CPU performance has begun slowing, and therefore vendors have turned to multicore processors to gain additional performance [Creeger, 2005; Geer, 2005]. This requires software developers to implement concurrency in the applications to take advantage of the additional cores, but it comes with certain challenges.

2.7.2 Race Conditions and Critical Sections

Even though threads run autonomously, they may be interleaving based on scheduling principles, and from a programmers perspective it happens at random. It can also happen because of external events, such as hardware interrupts or one thread depends on a input, while another does not. Because of these issues it is difficult to reason about where a thread is in its sequence of actions. This is an issue when the threads has to cooperate or use shared variables. Consider the case `x = x + 1`, where `x` is a shared variable, which consists of three operations: Load x, add 1 to x, store x. Listing 2.2 shows this example with two threads executing it in parallel with the result, that after one execution by both threads `x` will only have been incremented with one. This is called a *race condition* and is described in "A taxonomy of race conditions"[3] as when the events of two programs conflict (e.g., one reads and the other writes the same memory location) and their execution order depends on how the threads are scheduled [Helmbold and McDowell, 1996]. Race conditions break the deterministic behavior of applications and is not acceptable and has to be solved for multi-threading to be of use.

Pseudocode

```
   //Thread 1                    //Thread 2
1  Load x                       Load x
2  Add 1 to x                   Add 1 to x
3  Store x                      Store x
```

Listing 2.2: Thread 1 and Thread 2 both adding 1 to x.

In the case mentioned above it is necessary to make the *critical sections* of the threads (`x = x + 1`) mutual exclusive. Critical section is defined by Dijkstra as: "*Critical in the sense that the processes have to be constructed in such a way, that at any moment at most one of the two is engaged in its critical sector*" [Dijkstra, 1965a]. Dijkstra presents a solution in "Solution of a Problem in Concurrent Programming Control" that works for N computers [Dijkstra, 1965b]. However, this uses busy waiting, which is the repeated check of whether a condition is true, and it is an undesired activity. The *semaphore* was created to solve these problems and it is a non-negative integer with two operations: P increases the value of the semaphore and V decreases

[3] More specifically, if a race affects only affect data, then it is a data race. However if the result of a race condition causes a thread to take a different path, then it is a control race, earlier referred to as a general race [Netzer and Miller, 1992].

the value. When the semaphore is zero and a thread invokes a V operation on the semaphore the thread is put to sleep and inserted into a queue until a different thread has made a P operation on the semaphore.

2.7.3 Deadlock Issues

The semaphore does not solve all the challenges though, as locking mechanisms introduce new challenges. *Deadlock* (called "the Deadly Embrace" by Dijkstra [Dijkstra, 1965a]) is described by Hansen as: "*A deadlock is a state in which two or more processes are waiting indefinitely for conditions which will never hold*" [Hansen, 1973]. The example in listing 2.3 presents a potential deadlock. It makes use of a binary semaphore, which is a semaphore that can only be one or zero. The deadlock can happen when `thread 1` takes `binSem1`, `thread 2` takes `binSem2` and none of them can now continue, because `thread1` waits for `binSem2`, which is taken by `thread 2`, and `thread 2` waits for `binSem1`, which is taken by `thread 1`. *Livelock* was introduced by Ashcroft and describes a case where two threads acts to the response of each other and therefore keeps passing the resource and neither gets to use it [Ashcroft, 1975]. Using semaphores in a system can also lead to starvation, where a thread is denied access to a resource indefinitely. Consider the allocation of a printer, where the policy is to print the smallest files first. This can lead to larger files never getting printed, if there is a steady flow of small files added to the print queue. Prioritizing of threads can also lead to this problem, as higher prioritized threads can prevent lower prioritized threads from executing.

Pseudocode

```
   //Thread 1                    //Thread 2
1  p(binSem1)                    p(binSem2)
2  p(binSem2)                    p(binSem1)
```

Listing 2.3: Deadlock example

2.7.4 Referential Transparency

The cases mentioned above deal with shared resources and the protection of these, and some issues are caused by missing *referential transparency*. Referential transparency can be described as: "*...the meaning of an expression is its value and there are no other effects, hidden or otherwise, in any procedure for actually obtaining it*" [Bird and Wadler, 1988]. In other words, with referential transparency it is possible to substitute an expression with its value without changing the behavior of the program, thereby eliminating side-effects. Without referential transparency the order of events matters, meaning that invoking the same function with the same parameters can result in different values. This is a challenge in a concurrent context, because in most cases it is unknown from the programmers perspective when a thread gets to run, and thereby the order of events. Therefore, delegating a function without referential transparency to a new thread increases the need for synchronization to force the correct flow of events. The example in listing 2.2, imagine them both as functions, are not referential transparent, as they change a global variable. For example, none of the methods return any value, so substituting the expression with the value does change the behavior of the program.

2.8. Concurrency in Scala

As mentioned in chapter 1, Scala was used to implement the COE, and was therefore also used in this thesis project. For this reason, this section introduces Scala in brief and describes how Scala can be used to solve the challenges presented in the previous section.

Scala is an acronym for Scalable Language and was first released in 2003 [Odersky, 2006]. It compiles to Java bytecode and is interoperable with Java. It is an object-oriented language with functional paradigms where every value is an object and every operation is a method call. The meaning of ”every value is an object“ is, that float and int for example are objects, as in figure C.1 in appendix C, whereas in Java they are not. This provides additional flexibility e.g. in a polymorphic setting. ”Every operation is a method call“ means that e.g. 1 + 2 is invoking the method + defined in the Int class. This provides Scala with additional functionality, whereas in Java it is not possible to define + in a class. Scala is also a functional language where every function is a value. As stated above, every value is an object so every function is also an object. It is therefore possible to pass functions as parameters to higher-order functions, making it first-class citizens.

Scala provides language constructs to apply *immutability*, which means that the state of an object cannot be changed after construction [Peierls et al., 2005]. Consider the process of turning a Java class into an immutable class in listing 2.4, which can be instantiated with `new ImmutableJavaClass("Something")`. In Scala this can be done much simpler using case classes as shown in listing 2.5 and the instantiation `ScalaCaseClass("Something")`. Scala does not enforce immutability, meaning that it is possible to declare variables using **`var`**, thereby making them mutable. However it does encourage immutability, e.g. parameters are immutable by default. It is important to mention, that immutability does not necessarily mean that data is copied every time it is passed to a function. Scala makes use of data sharing, which is possible because the data is immutable, so it is not necessary to copy it, but instead it can be reused [Chiusano and Bjarnason, 2014, p. 35-37]. Immutability is an important attribute of functional programming

Java

```
1  public class ImmutableJavaClass
2  {
3    //The final keyword means immutableVar can only be
       ↪ initialized once
4    final private String immutableVar;
5
6    public Foo(final String initialValue)
7    {
8      this.immutableVar = initialValue;
9    }
10
11   public String getVar()
12   {
13     return this.myvar;
14   }
15 }
```

Listing 2.4: An immutable Java class ImmutableJavaClass

```Scala
case class ScalaCaseClass(immutableVar: String)
```

Listing 2.5: A case class in Scala

and it addresses the challenges mentioned in section 2.7 as described below.

Race condition: As specified in section 2.7 a race exists when two programs conflict in the sense that one reads and another writes the same memory location. Using Scala's immutability principles this cannot happen with pure functions. A pure function is a function that always evaluates to the same result given the same input values, and thereby does not depend on any hidden information or external input, and the function does not cause any semantically observable side-effect. Therefore, a pure function is also referential transparent. The expressions in listing 2.2 are not possible, because `x` is immutable and can therefore not be changed once defined.

Locks/Deadlocks: Keeping the data immutable and using pure functions avoids the need for locks, as it is not necessary to protect regions to prevent changes in variables.

Referential Transparency: Referential transparency is an attribute of functional programming, however it is not limited to functional programming languages. By ensuring referential transparency it is straight-forward to delegate a function to a different thread, because it does not depend on anything else than its parameters and can therefore do its processing unaware of the system around it. Keeping the data immutable also supports referential transparency, as there are no variables to alter.

Immutability and pure functions are useful, but for a program to be useful it is often necessary to communicate between threads, use external resources such as the file system and use other applications, thereby introducing shared resources and impure functions. However emphasizing immutability and purity in programming reduces the area of code susceptible to race conditions, deadlocks and other concurrency challenges.

2.8.1 Futures

A *future* is a placeholder for a value, that is the result of some concurrent calculation, and it can be accessed synchronously or asynchronously. The term future was proposed by Baker and Hewitt:

> "*In call-by-future, each formal parameter of a function is bound to a separate process (called a 'future') dedicated to the evaluation of the corresponding argument.*" [Baker and Hewitt, 1977]

It is similar in nature to the term *promise* used by Friedman and Wise in relation to lazy evaluation of a data structure [Friedman and Wise, 1976, page 268], where a variable is bound to the promise of a result, which will not be computed until required. Hibbard used the term *eventual value* which, like futures, is used for parallel processing [Hibbard, 1977, page 1–7]. The name refers to the real value of a parallel execution eventually becoming available, but upon initial assignment of an eventual value to a variable it is not the real value.

A future has two possible states: undetermined or determined. Initially it is undetermined, and when a value is set in the future object, it becomes determined. A future cannot contain more than one value, and it cannot be determined more than once. Futures are considered to be first-class citizens in the sense that they can be passed as arguments, returned from functions, and stored in data structures [Katz and Weise, 1990]. The future construct causes the following operations to be performed:

- When an expression is passed, a future is returned. This future is a promise to deliver a value, once the value of the expression is resolved.
- A new thread is created to evaluate the expression.
- Once the created thread finishes evaluating the expression, the value is made available.

To shift from the general future concept to futures in Scala, it is necessary to present a few definitions:

- A determined future corresponds to a completed future in Scala.
- An undetermined future corresponds to not completed future in Scala.
- The use of futures above referred to the future concept in general. From this point forth, future will refer to futures in Scala. When the specific type/code instance of a future is referred to, it is capitalized and written in typewriter font as such: `Future`.

To run several concurrent processes it can be necessary to compose futures. Scala provides a function `sequence`, which turns an iterable collection of future constructs into one future construct as shown in listing 2.6; in general, the function `sequence` turns a subclass `I` of `Iterable`, `I[Future[A]]` into `Future[I[A]]`. This makes it possible to for example wait till all concurrent processes are finished, as shown on line four in listing 2.6, where a blocking wait is performed in line three.

Scala

```
//getFutures is a function generating an iterable
   ↪ collection of Future<Integer> objects
1 Iterable<Future<Integer>> listOfFutureInts = getFutures()
2 Future<Iterable<Integer>> futureListOfInts = sequence(
   ↪ listOfFutureInts)
3 Await.ready(futureListOfInts, Duration(5,TimeUnit.SECONDS))
```

Listing 2.6: Futures sequence

A future can result in failure or success once completed, as shown in figure 2.10. It is possible to provide callback functions for both failures and successes. Listing 2.7 shows a success callback function, along with a completion callback function using pattern matching. A callback function in this case is a function, that is executed once the future results in failure or success. Pattern matching tries to match a value against several patterns, with each pattern pointing to an expression. The expression associated with the first matching pattern will be executed.

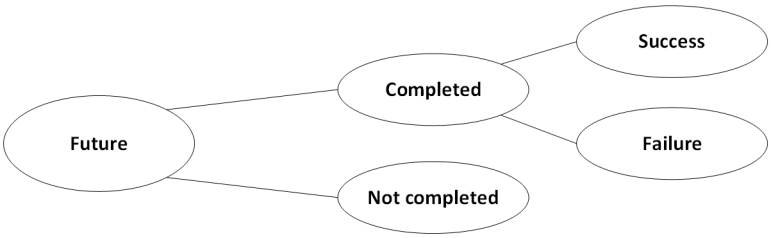


Figure 2.10: A future is either completed or not completed; once completed the result can be a success or failure.

Scala

```
1 val originalFuture = Future {getResult()}
2 originalFuture onSuccess {case result =>
3   val actionOnResult = Future {
4     performActionOnResult(result) }
5   actionOnResult onSuccess {case _ => println("Success")}
6 }
```

Listing 2.7: `OnSuccess` callbacks

The example in listing 2.7 can be rewritten using the combinator `map` as shown in listing 2.8. A combinator is a higher-order function that only depends on its inputs, which includes a function. The `map` combinator, given a `Future` and a function, produces a new `Future` that is completed with the value of the function applied to the value of the original successfully completed future. This rewriting helps to avoid nesting callback functions and thereby avoiding closures, such as the closure inside the `OnSuccess` callback function on line two in listing 2.7. The `actionOnResult Future` is therefore available at the same scope as the rest of the code, making it possible for other parts of an application to register callback functions. Another way

Scala

```
1 val originalFuture = Future {getResult()}
2 val actionOnResult = originalFuture map {result =>
3   performActionOnResult(result) }
4 actionOnResult onSuccess {case _ => println("Success")}}
```

Listing 2.8: Future map combinator

of composing futures is using a for-comprehension as illustrated in listing 2.9. This performs two `doSteps` in parallel and validates the steps. If the steps are valid, the result is retrieved. This example can be translated into listing 2.10. The `flatMap` function on line three is necessary to avoid nested futures.

Futures require an `ExecutionContext`, which is responsible for executing computations. An `ExecutionContext` is an abstraction for working with threads, and it can execute the computations in a new thread, a pooled thread, or in the current thread. It is possible to implement your own threading strategy or use a predefined strategy, where it is possible to set parameters such as pool size, which is a measure of how many threads to use.

Scala

```
1 val fmu1Future = Future {doStep(fmu1)}
2 val fmu2Future = Future {doStep(fmu2)}
3 val results = for {
4   fmu1Step <- fmu1Future
5   fmu2Step <- fmu2Future
6   if (validateSteps(fmu1Step, fmu2Step))
7   } yield getResults(fmu1Step, fmu2Step)
```

Listing 2.9: Futures for-comprehension

Scala

```
1 val fmu1Future = Future {doStep(fmu1)}
2 val fmu2Future = Future {doStep(fmu2)}
3 val results = fmu1Future flatMap => {
4   fmu1Step => fmu2Future
5   .withFilter(fmu2Step => validateSteps(fmu1Step,fmu2Step))
6   .map(fmu2Step => getResults(fmu1Step, fmu2Step))
7   }
```

Listing 2.10: Translation of the for-comprehension in listing 2.9

2.8.2 Actors in Scala

In 1973 the concept of the *actor model* was introduced in the context of artificial intelligence [Hewitt et al., 1973]. It was introduced as an architecture to efficiently run programs with a high degree of parallelism without the need for semaphores. Erlang, "*a language designed for writing concurrent programs that 'run forever'*" [Armstrong, 2007], is using the actor model and has been a popular language for developing parallel systems. Erlang's success has also influenced the development of the Scala Actors library: "*Our library was inspired to a large extend by Erlang's elegant programming model.*" [Haller and Odersky, 2009]. Furthermore, Agha and Kim argues, that the actor model is suitable for developing software for real-world systems, because it is capable of handling both parallel and distributed computing [Agha and Kim, 1999]. The difference between distributed computing and parallel computing in this context is, that parallel computing assumes reliable communication links and that processes are "close" to each other, so communication is faster and more trustworthy than in distributed computing, where the processes are dispersed in a wide area and networked. They argue, that it is important to be able to handle both distributed and parallel computing, because there is a trend towards a convergence of these as network technology improves, which Haller and Odersky agrees with in the article "Scala Actors: Unifying thread-based and event-based programming" [Haller and Odersky, 2009]. The actor model provides an abstraction to concurrency and distributed systems. An *actor* is an autonomous object and encapsulates the following [Agha and Kim, 1999]:

Data: Data can be encapsulated inside an actor. This also helps to control state, if it is necessary. It is important that state is not shared, as it would complicate synchronization.

Methods: An actor can contain several methods.

Thread: To prevent synchronization issues an actor encapsulates a thread, which also makes it autonomous.

Mailbox: An actor has an ordered mailbox, in which incoming messages are stored.

Mail address: This is a globally unique reference for an actor.

"*An actor is a concurrent process that communicates with other actors by exchanging messages*" [Haller and Odersky, 2009]. The communication is asynchronous and each message is buffered in a mailbox. This is race-free by design and therefore it is possible to avoid synchronization mechanisms. The processing of a message is done by the actor without interruptions, which makes it atomic in nature, and that is vital to avoid synchronization problems as mentioned above, because an actor might keep internal state. If messages are handled in parallel, then synchronization becomes necessary to prevent undesired behavior based on state, which was described in section 2.7. The basic reactions of an actor upon receiving a message is: Send messages to other actors, spawn new actors, or make changes to its local data [Agha and Kim, 1999].

This thesis will deal with Akka's implementation of actors and not Scala Actors[4] [Typesafe Inc, 2015], and therefore actors will refer to Akka actors from this point forth. When the specific type/code instance of an actor is referred to, it is capitalized and written in typewriter font as such: `Actor`.

One or more actors that share services such as logging, configuration etc., are referred to as *actor systems*. It is possible for several actor systems with different configurations to co-exist, making the actor model versatile, because it is possible to have different kind of actors. As actors can spawn other actors, it is possible to create hierarchical structures. An actor spawning other actors is called a *supervisor-actor*, and the spawned actors are called *child-actors*. It is possible for the supervisor-actor to monitor and provide fault handling for its child-actors. The underlying details of communication with other actors are abstracted away, such that communication with remote actors is no different from application perspective than communication with local actors. The configuration of actors is done via the actor system, and besides threading strategies it is also possible to configure mailboxes, logging, deployment of remote actor systems, routing etc.

Using actors a message, `msg`, can be sent to an `Actor` instance, in this example named `actor`, with "Fire-Forget" by writing `actor ! msg`. Receiving messages is based on pattern matching as shown in listing 2.11. If there are no patterns matching a given `msg`, then it is send to a synthetic actor, making `receive` a partial function.

Scala

```
1 receive : PartialFunction[Any, Unit] = {
2   case message1 => method1Action
3   case message2 => method2Action
4   ...
5 }
```

Listing 2.11: Receive syntax in Scala actor

It can be seen in listing 2.11 that the type of the receive method in an actor is `PartialFunction[Any,Unit]`. According to figure C.1 in appendix C `Any` means, that the receive function

[4]This is because the proposed `Actor` library for Scala was changed in Scala 2.11.0, see `http://docs.scala-lang.org/overviews/core/actors.html`

can take any type as input, `Unit` means there is no return value, and `PartialFunction` means the method is not defined for all messages types . As mentioned above, the sending of messages uses references to actors, meaning an actor can have been deleted, but the sending will still appear as a success to the sender. Using `?`, which is called "Ask" or "Send-And-Receive-Future", instead of `!`[5] when sending a message to an actor, results in the sender receiving a `Future` object, which was described in section 2.8.1.

There is an alternative to regular actors called *typed actors*. Typed actors makes it possible to avoid the `PartialFunction`, the `Any` type and by the `Unit` type as well. With typed actors a static contract is used and it is therefore not necessary to define messages, as it is with pattern matching. Using typed actors will help bridging between regular Object-Oriented code and actor systems. It is possible to use both "Send-And-Receive-Future" and "Fire-Forget" with typed actors.

2.8.3 Parallel Collections

Scala also provides *parallel collections* as a simple high-level abstraction to parallel programming. Many applications use functions, such as map and filter, that operates on data structures like lists and hashtables. These functions are usually performed sequentially because they depend on iterators. The article "A Generic Parallel Collection Framework" [Prokopec et al., 2011] describes an approach to parallelizing collection operations in a generic way, which was used for implementing parallel collections in Scala. To support multiple processors the work is distributed among processors with each processor maintaining a queue. If a processor is done with all tasks in its queue, it can *steal* tasks from another processor's queue. To schedule tasks the Java fork-join framework is used, as shown in listing 2.12, which is a divide-and-conquer algorithm [Lea, 2000]. The fork-join framework partitions a task into smaller tasks based on a threshold, in the case of Scala Parallel Collections: $threshold = max(1, n/8P)$, where n is the number of elements to process and P is the number of processors.

Pseudocode

```
1 Result solve(Problem problem){
2   if (problem is small)
3     directly solve problem
4   else {
5     split problem into independent parts
6     fork new subtasks to solve each part
7     join all subtasks
8     compose result from subresults
9 }}
```

Listing 2.12: Fork/join algorithm [Lea, 2000]. Fork starts a new parallel subtask, join causes the current task not to proceed until the forked subtask has completed.

To parallelize a collection, the function `par` must be invoked on the sequential collection, as shown in listing 2.13. After par has been invoked on the collection, the collection can be used the usual way. The example in listing 2.13 adds 42 to every element of a list in parallel. The value `parList` on line two has the type `ParSeq[Int]`, which makes it possible to define a threading configuration. It would also be possible to convert the collection to a `par` collection

[5]The `?` and `!` notation has its origin in Communicating Sequential Processes [Hoare, 1985].

on line one by writing `val list = (1 to 10000).toList.par` and leaving out `par` on line two. It is possible to configure e.g. the thread pool size on parallel collections, but the type must be converted first, for example using `par` as mentioned above. It is important to notice, that the `par` function converts the collection into a parallel collection, and when e.g. a `map` function is performed, the result is also a parallel collection, making future functions performed on the collections execute concurrently. This might not be desirable, and therefore the function `seq` can be used to convert the parallel collection a regular sequential collection.

Scala

```
1 val list = (1 to 10000).toList
2 val parList: ParSeq[Int]: list.par.map(_ + 42)
```

Listing 2.13: Parallelized map operation

Chapter 3

Co-Simulation Orchestration Engine Implementation

This chapter describes the baseline implementation of the Co-Simulation Orchestration Engine (COE) after it was established. The baseline performs Co-Simulation using the Functional Mock-up Interface (FMI), as described in chapter 2. Additionally, this chapter also presents the refactored version of the baseline and the three concurrency implementations using the concurrency features, that was presented in chapter 2. Later in this thesis, these implementations are tested and evaluated to confirm or refute the hypothesis. The results of the testing and evaluation is described in chapter 4. Based on the testing, evaluation, and knowledge gained by performing these implementations, this chapter helps to fulfill the entirety of goal 3: Learning Scala and different concurrency features and additionally, providing knowledge for generalizing on implementing functionality that uses concurrency in an application for chapter 5. Because these implementations also perform Co-Simulation using FMI, this chapter also helps to achieve goal 1.

3.1. Introduction

Chapter 3 contains the realization of the Co-Simulation Orchestration Engine (COE) and how concurrency was implemented in the application using the concepts mentioned in chapter 2. As described in the approach in chapter 1 the following development tasks had to be performed: Establishing a baseline, refactor the baseline and add concurrency features to the refactored baseline. Most of the baseline implementation was done prior to this thesis, and only some minor refactorings and bug-fixes were performed as part of this thesis project to establish a baseline, and are therefore not described. After the baseline based on the Master Algorithms (MAs) described in chapter 2 was established, an implementation using actors was performed without refactoring the baseline, so it was possible to see whether the refactoring had any effect. Then the baseline implementation was refactored to enable an optimized implementation of functionality that uses concurrency. Next, three different implementations were performed on the refactored baseline using the following concurrency features: parallel collections, futures, and actors.

The baseline implementation performing Co-Simulation using the Functional Mock-up Interface (FMI) is described in section 3.2 along with details on how to run simulations. Afterwards, sec-

tion 3.3 describes the refactoring that was performed to the baseline, in order to prepare it for implementing concurrency. It also describes the concurrency implementations making use of futures, actors, and parallel collections as described in section 2.8.

3.2. Co-Simulation Orchestration Engine Baseline Implementation

In this section the baseline implementation of the COE is described. The implementation was carried out by Kenneth Lausdahl, Aarhus University, with assistance from Victor Bandur, Aarhus University, and the author of this thesis. The following section, section 3.2.1, presents how to run simulations using the COE application, which also applies to the concurrent implementations. Afterwards Section 3.2.2 describes how the simulation phase is performed in the baseline implementation.

3.2.1 Co-Simulation Orchestration Engine Server

The COE application runs as a web server, and will be referred to as the COE server in this section to distinguish it from clients. The idea is to make it available for several clients and use sessions to separate them, using the session ID as a key. This section does not contain the full server protocol and options, but only the parts used in this thesis; for more information see the ”protocol.pdf“ file located within the CD. The COE server invocation order consists of three HTTP requests, which can be seen in figure 3.1.

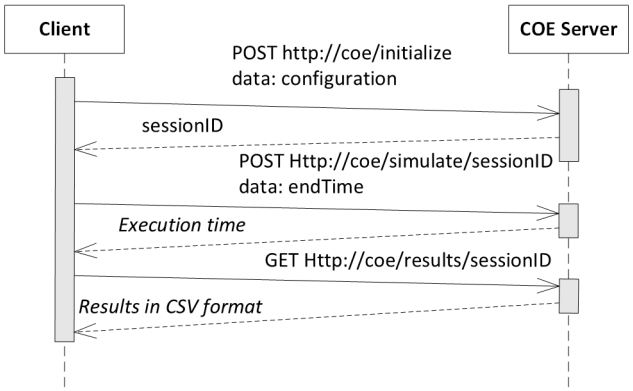


Figure 3.1: Server requests for performing a simulation and retrieving the results.

These requests consists of POST and GET requests, where a POST request to a web server is a request that can contain data and therefore submits data, whereas a GET request to a web server requests data. The requests are described below:

Initialize: In this step the COE configuration file is sent as a POST request to the server via the URL `http://coe/initialize`. The COE configuration file describes the entire model, an example of a model is shown in figure 2.3 in chapter 2, meaning it contains the Functional Mock-up Units (FMUs), the mappings between the outputs and inputs of the FMUs, optional parameters and a MA. The structure of the COE configuration file can be seen in listing 3.1. Once the COE configuration file has been sent to the COE server, the

COE is initialized and associated with a session ID, the FMUs are loaded, and a fixed or variable step size algorithm is chosen. The session ID is returned, so it can be used in future requests to locate the correct COE session.

Simulate: This step is a POST request to the URL `http://coe/simulate/sessionID`. The session ID is used to locate the COE session initialized in the previous step and use it to run the simulation, which was also specified in the previous step. Among other data the simulation execution time is returned.

Results: To get the result of a simulation it is necessary to send a GET request to the URL `http://coe/results/sessionID`. This request returns the result of the simulation in a CSV formatted string containing time, stepsize, and all outputs.

JSON

```
{
  "fmus": ["fmuPath1","fmuPath2"],
  "connections": {"guid.outputVariable": ["guid1.
    inputVariable","guid2.inputVariable"]},
  "parameters": {"par1": value, "par2": value2},
  "algortihm": {
    "type": "fixed-step",
    "size": 0.1
  }
}
```

Listing 3.1: Structure of simulation configuration file

3.2.2 Simulation Phase

The simulation initiated by a POST request to the URL `http://coe/simulation/sessionID` contains several parts. A state diagram containing the overall simulation process can be seen in figure 3.2. `ModelInstance` is a class in Scala representing an FMU and will be used throughout this chapter. The steps on the figure are separate functions, meaning that multiple mappings are performed over the collection of `ModelInstances` in a single simulation step. The parts of the simulation process that are relevant for concurrency are described below:

Resolving inputs: As shown in listing 3.1 the configuration file contains the mappings from output to inputs in the connections object. These are added to two `Maps`: `Inputs = Map[ModelInstance, Map[ScalarVariable, Tuple2[ModelInstance, ScalarVariable]]]` and `Outputs = Map[ModelInstance, Set[ScalarVariable]]`. By looking at the types it can be seen, that `ModelInstance` is being used as key. `Inputs` are ”self-contained“ in the sense, that it contain references to both the `ModelInstance` on which to set the input, but also the `ModelInstance` from which to retrieve the output.

Set inputs: After the inputs have been resolved, they are set on the FMUs. To set them on the FMUs another `Map` with `ModelInstance` as key and a reference to the actual FMU as

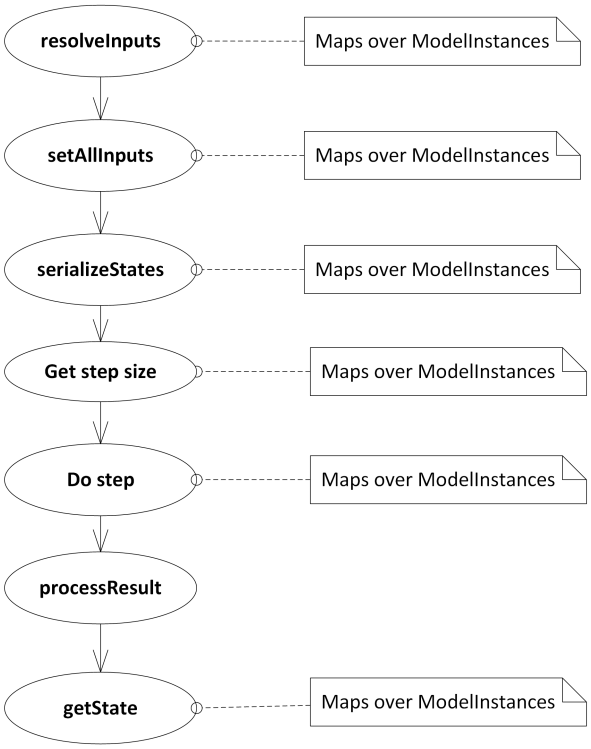


Figure 3.2: The overall process of a simulation step in the baseline

value is used. The `ModelInstance` thereby serves as key in both outputs, inputs and instances.

Serializing state: If the FMUs are capable of getting and setting state, then the state is serialized and retrieved from the FMU to enable rollback. This is done after setting the inputs to avoid having to set the inputs again in case of a rollback.

Get stepsize: The COE supports two types of steps: fixed and variable step size. In case of variable step size it is necessary to use the function `fmi2GetMaxStepSize` proposed in section 2.5.3. Using fixed step size, the step size configured in ”initialize“ described in section 3.2.1 is used.

Do Step: The `doStep` method is invoked on the FMUs, which returns a status of either `OK`, `Error`, `Fatal`, `Discard` or `Pending`.

Process result: Based on the status returned from performing a `doStep` different actions are taken. `Error` and `Fatal` both causes the COE to terminate. `Pending` means the FMU is performing the `doStep` asynchronously, which also causes the COE to terminate. If an FMU returns `Discard` then it was unable to perform the step, however it might have completed part of it. The FMUs returning `Discard` are then invoked to find out how much of the step they were able to complete. Based on the minimum value of the step the FMUs were able to complete, they are rolled back and rerun with this step size.

Retrieving the new state: Before running the next step, the new state after the `doStep` is retrieved. In case of the COE using variable step size the new state is validated, and based on the validation a rollback is performed. Once the new state is retrieved, the simulation can continue with the next step.

3.3. Co-Simulation Orchestration Engine Concurrency Implementation

There are several ways to apply concurrency to an application and this section describes the different implementations that were realized. The implementations are different ways of performing some of the tasks an MA must perform, see figure 2.6 in chapter 2 and figure 3.2. This means that the implementation of ”Initialization“ and ”Tear down“ in figure 2.6 has not been changed. The implementations were developed separately, so an implementation that changed something else than the MA implementation did not have an impact on the other implementations. Section 3.3.1 describes a refactoring of the MA in the COE application, that forms the basis for the final implementations of concurrency in the COE: parallel collections in section 3.3.2, futures in section 3.3.3, and actors in section 3.3.4. Finally, section 3.3.5 describes an implementation, where the baseline was minimally changed with some FMU function calls optionally being realized in an actor.

3.3.1 Refactoring to Limit Synchronization

When implementing concurrency in the COE application it is desirable, that as much work as possible is performed concurrently without having to synchronize between threads, as synchronization takes time. Synchronize in this context means waiting for the threads to finish, so values can be retrieved and set safely. This section describes the refactoring performed to the baseline to better support maximizing the workload performed concurrently. Considering the baseline implementation, the flow is shown in figure 3.2, in a concurrent perspective, where the functions `setAllInputs`, `seralizeStates`, `doStep`, possibly rollback in `processResult`, and `getState` are invoked concurrently for each FMU. This will contain many synchronizations as illustrated in figure 3.3, where the COE application must wait for all threads to finish in order to continue. It is also clear, that there are many mapping operations which all map over the `ModelInstances`. The goal of the refactoring process is to limit the synchronization and the mapping operations. Synchronization cannot be completely eliminated, because it is necessary to resolve the inputs for the FMUs before every step, which requires the outputs from the FMUs, and the simulation cannot continue until this is performed. However, it is still possible to perform a refactoring, where more work is done between synchronizations.

The refactored implementation is shown in figure 3.4. This effectively reduced the mapping operations from six, possibly seven depending on `processResult`, to three and synchronizations from four, possibly five depending on `processResult`, to one. Previous to the refactoring, the COE application sequentially controlled the entire flow by calculating the necessary parameters for the next function to invoke on the FMUs, invoking one function on the FMUs and so on until a simulation step was performed. After the refactoring, functions in the COE application directly related to invoking functions on the FMUs are encapsulated into one entity, and thereby separated from the calculation of the necessary parameters. This transforms the flow to: Calculate the necessary parameters for an entire simulation step, invoke the encapsulated and separated entity, which performs the entire simulation step. This encapsulated entity will be referred to as the concurrent entity, and represents the frame in figure 3.4. By having this functionality separated and encapsulated, it is straightforward to wrap it in a thread, that performs more work. This is related to immutability and referential transparency, which was described in section 2.8. However it comes with a trade off: In case one or more FMUs fail in the `doStep`, the state would not be retrieved previous to the refactoring. After the refactoring, the state of the FMUs not failing in the

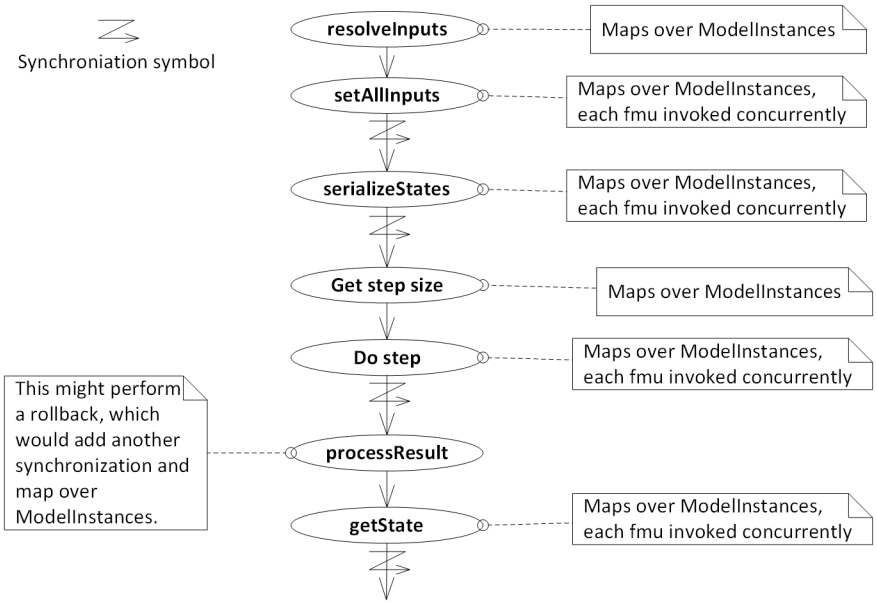


Figure 3.3: The overall process of a simulation step in the baseline implementation with synchronization.

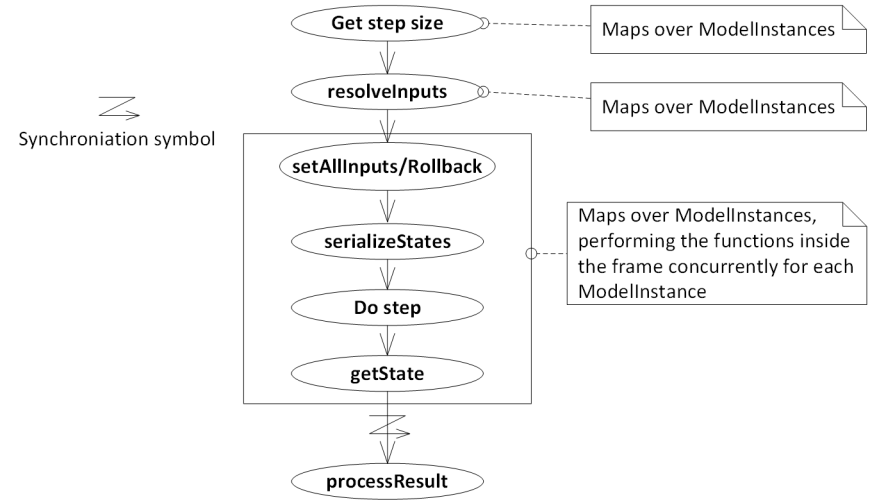


Figure 3.4: The overall process of a simulation step in the concurrent implementation with synchronization.

`doStep` process would still be retrieved, because the entities responsible for the FMU simulation step are unaware of the state of other entities until the synchronization phase. This might lead to unnecessary retrieval of states.

As mentioned in section 3.2.2, the COE application contains a `Map` called `instances` with the `ModelInstances` as keys and their related FMUs as values. This is used for invoking concurrent entities as in listing 3.2, where a concurrent entity is wrapped in the function `doSimulationStep`.

Scala

```
1 val doStepResults = instances.map {
2   case (mi, si) => {
3     doSimulationStep(mi, si, currentCommunicationPoint,
        ↪ communicationStepSize, resolvedInputs.get(mi),rb)
4 }}
```

Listing 3.2: Iterating over instances and invoking concurrent entities with the function `doSimulationStep`.

3.3.2 Implementation using Parallel Collections

This section describes how parallel collections were used to implement concurrency in the COE. As described in section 2.8.3, parallel collections are a simple high-level abstraction to parallel programming. As the refactoring described in section 3.3.1 separated and encapsulated the functionality to be wrapped in a thread, it was a conceptually simple task to take advantage of parallel collections. As a collection is converted to a parallel collection by adding `par`, which was described in section 2.8.3, it is possibly to make the implementation in listing 3.2 concurrent by adding `par`, as shown in listing 3.3. When performing functions on the resulting parallel collec-

Scala

```
1 val doStepResults = instances.par.map {
2   case (mi, si) => {
3     doSimulationStep(mi, si, currentCommunicationPoint,
        ↪ communicationStepSize, resolvedInputs.get(mi),rb)
4 }}
```

Listing 3.3: Using parallel collections to concurrently invoke the concurrent entities.

tion, it must be determined whether these functions should be run in parallel or not. As mentioned in section 2.8.3, the result of a function performed on a parallel collection is also a parallel collection. The function `seq` is therefore used on the `doStepResults` value to convert it to a sequential collection for further processing.

3.3.3 Implementation using Futures

In this section it is described how futures were used to implement concurrency in the refactored COE application described in section 3.3.1. Recall, that a future is a placeholder for a value, that is the result of some concurrent calculation and can be accessed synchronously or asynchronously as described in section 2.8.1. Invoking the concurrent entities using futures is shown in listing 3.4, which also makes use of `instances`, similar to the implementation using parallel collections in section 3.3.2.

It is necessary to access all the FMUs for processing the results, getting the next step size and resolving inputs as illustrated by the synchronization symbol in figure 3.4, and therefore the futures are accessed synchronously. The function `Await.result` on line five is similar to `Await.ready` described in section 2.8.1, but besides waiting for the future to be completed it

Scala

```
1 val doStepResultsFuture = instances.map{
2     case (mi, si) => { Future {
3     doSimulationStep(mi, si, currentCommunicationPoint,
        ↪ communicationStepSize, resolvedInputs.get(mi),rb)
4 }}}
5 val doStepResults = Await.result(Future.sequence(
    ↪ doStepResultsFuture), Duration(5, TimeUnit.SECONDS))
```

Listing 3.4: Using Futures to concurrently invoke the concurrent entities.

also retrieves the result. The `Await.result` function takes a `Future[sometype]` type, and since the type of `doStepResultsFuture` is `Iterable[Future[doSimulationStepResult]]`, it is necessary to transform the type into `Future[Iterable[doSimulationStepResult]]`. This conversion is made possible by the function `Future.sequence` on line five, and thereby the `doStepResults` value contain the results of the simulation step invoked on all the concurrent entities and can be processed synchronously. The last part of the future implementation is adding an `ExecutionContext` responsible for controlling the threading strategy, as it is required for using futures. This implementation uses the default `ExecutionContext` by **`import`** `ExecutionContext.Implicits.global`.

3.3.4 Implementation using Actors

The implementation of the refactored COE application described in section 3.3.1 using actors is described in this section. An actor, described in section 2.8.2, is an autonomous object encapsulating data, methods, a mailbox and has an address. The usage of actors requires more changes than the usage of parallel collections and futures. The first step was to create an actor system, which is used to create and instantiate `Actor` objects as shown in listing 3.5 in line 1.

Scala

```
1 val system = ActorSystem()
2 val instances = ModelInstances.map{ mi => mi ->
3   val actor: FmuActor = TypedActor(system).typedActorOf(
      ↪ TypedProps(classOf[FmuActor], new FmuActorImpl(mi,
4   comp, instanceConfig)))
5   new FmiSimulationInstance2(comp, instanceConfig, actor)
6 }.toMap
```

Listing 3.5: Using Futures to concurrently invoke the concurrent entities.

Recall from sections 3.3.1 to 3.3.3 that the `instances` value was used as a starting point for concurrency. As an actor exists throughout the entire simulation and it is desirable to keep as much of the implementation performed in the refactoring as possible to increase reuse, the contents of the `instances` value was changed, which can be seen in line three to six in listing 3.5. Typed actors are used in the implementation in order to have a static interface, which makes for better type checking as described in section 2.8.2. `FmuActor` is the trait, similar to an interface, for the

actor and can be seen in appendix D.

Actors encapsulate a thread and methods, and therefore the function `doSimulationStep` used to invoke the concurrent entities was moved to the `Actor` objects. Because of this and with the change to the `instances` value, the concurrent invocation of the simulation step now looks as the implementation shown in listing 3.6. The type of actor invocation used in this implementa-

Scala

```
1 val doStepResultsActor = instances.map {
2   case (mi, si) => {
3     si.actor.doSimulationStep(mi, si,
        ↪ currentCommunicationPoint, communicationStepSize,
        ↪  resolvedInputs.get(mi), rb)
4 }}
5 val doStepResults = Await.result(Future.sequence(
    ↪ doStepResultsActor), Duration(5, TimeUnit.SECONDS))
```

Listing 3.6: Using Futures to concurrently invoke the concurrent entities.

tion is ”Send-And-Receive-Future“, which means the function `doSimulationStep` returns a `Future`. Therefore it is possible to wait for the actor to finish the concurrent simulation step in the same way, as in the implementation using futures described in section 3.3.3. Because of this similarity, line five in listing 3.6 was already described in section 3.3.3 and is not described here.

3.3.5 Actor Implementation: Multiple Actor Invocations

The multiple actor invocations implementation is based on initial testing of implementing concurrency using actors, described in section 2.8.2. The structure is similar to the baseline simulation flow in figure 3.2, but with the difference that every FMU is also wrapped in an actor. This has changed the process of a simulation step from the baseline flow in figure 3.2 to the flow shown in figure 3.5. The setup of actors in this implementation is the same as described in section 3.3.4 and shown in listing 3.5 The default actor configuration was used, and the `Actor` trait in listing D.1 shows the functions encapsulated in the actor. In this implementation it is possible to pass arguments to the COE server application upon launching it and these arguments control the level of concurrency and thereby the invocation of the `FmuActor` functions in listing D.1. The arguments and related functions are described below:

NONE/nothing: If `NONE` is passed or no argument is passed to the COE application then the COE does not make any use of concurrency and executes similar to the baseline implementation.

GETSTATE: If `GETSTATE` is passed as an argument, the correct `get`-functions are executed on the FMUs concurrently. This is performed in the `getState` function of the actor.

SETVARIABLES: If `SETVARIABLES` is passed as an argument, the correct `set`-functions are executed on the FMUs concurrently. This is performed in the `SetInputVariables` of the actor.

DOSTEP: If `DOSTEP` is passed as an argument, the `doStep` function is executed on the FMUs concurrently. This is performed in the `doStep` function of the actor.

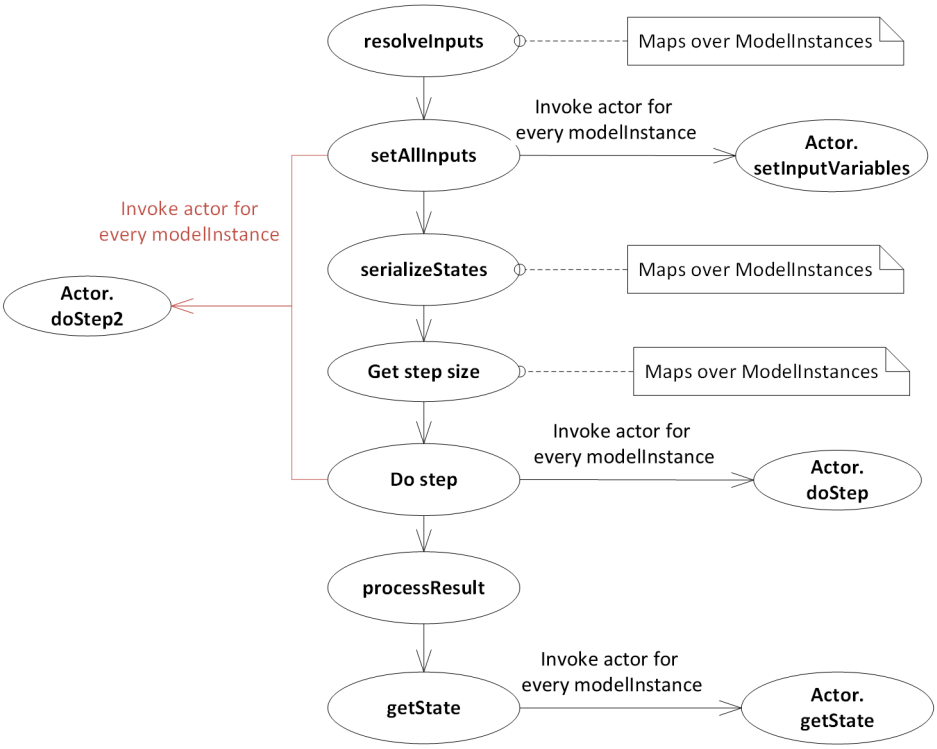


Figure 3.5: Simulation step flow in the multiple actor invocations implementation. The red line indicates an alternative flow, where `setAllInputs` and `doStep` are merged into a single function call, `doStep2`, to the `actor`.

Scala

```
1 trait FmuActor {
2   def doStep2(currentCommunicationPoint: Double,
      ↪ communicationStepSize: Double,
      ↪ noSetFMUStatePriorToCurrentPoint:
3   Boolean, inputState: InputState): Future[Fmi2Status]
4   def doStep(currentCommunicationPoint: Double,
      ↪ communicationStepSize: Double,
      ↪ noSetFMUStatePriorToCurrentPoint:
5   Boolean): Future[Fmi2Status]
6   def setInputVariables(inputState: InputState): Future[
      ↪ Unit]
7   def getState(getDerivatives: Boolean = true): Future[
      ↪ FmuActorState]
8 }
```

Listing 3.7: FmuActor trait

DOSTEPSETVARIABLES: If `DOSTEPSETVARIABLES` is passed as an argument, the functions `DOSTEP` and `SETVARIABLES` are executed on the FMUs concurrently. This is performed in the `doStep2` function of the actor, and is represented by the red line in figure 3.5.

ALL: If `ALL` is passed as an argument, it is the same as passing `GETSTATE`, `DOSTEP`, and

SETVARIABLES.

It is possible to pass any subset of GETSTATE, DOSTEP, SETVARIABLES, but it will perform as separate invocations of the actor functions. The change of behavior is based on arguments and handled using conditionals as exemplified in listing 3.8. This shows, that if DOSTEPSETVARIABLES is set, then an entirely different implementation no shown in the listing is used, if ALL or SETVARIABLES is set, then the Actor is used, and otherwise the baseline implementation is used.

Scala

```
1  def setAllInputs(instances: Map[ModelInstance,
       ↪ FmiSimulationInstance2],
2  resolvedInputs: Map[ModelInstance, InputState]): Unit = {
3    if (concurrent.contains(concurrentEnum.DOSTEPSETVARIABLES
       ↪ ) == false) {
4      if (concurrent.contains(concurrentEnum.ALL) ||
         ↪ concurrent.contains(concurrentEnum.SETVARIABLES))
5      {
6        val futures: Iterable[Future[Unit]] = resolvedInputs.
           ↪ map { case (modelInstance, inputState) =>
7          instances(modelInstance).actor.setInputVariables(
             ↪ inputState)
8        }
9        Await.ready(Future.sequence(futures), Duration(5,
           ↪ TimeUnit.SECONDS))
10     } else {
       //baseline implementation
11   }
12 }
```

Listing 3.8: Example of argument processing in setAllInputs.

Chapter 4

Evaluation of Implementations

This chapter describes how the concurrent implementations described in chapter 3 are tested and evaluated. The three different concurrency implementations were tested versus the baseline and each other. Several tests were implemented and compared to provide results for analyzing the implementations. To either confirm or refute the hypothesis, the performance results were compared and analyzed. Besides comparing the implementations based on performance, they were also compared on other quality attributes. This can be useful for future work, which is described in chapter 6. Furthermore, the testing and evaluation results are used to generalize the usage of concurrency in the Co-Simulation Orchestration Engine (COE) presented in chapter 5. Therefore, this chapter also helps to accomplish goal 3, where a part of the goal was generalizing on implementing functionality that uses concurrency in an existing application.

4.1. Introduction

This chapter presents the evaluation and verification of the concurrent implementations of the COE described in chapter 3. As part of the evaluation and verification was carried out by automatic testing, it was useful to define principles, that the testing should adhere to in order to get usable results. To enable automatic testing a framework was developed. This enabled testing of different concurrent implementations, evaluation of performance, and verification of consistency between the baseline simulation results and the concurrent simulation results. Some automated tests were already implemented prior to this thesis project, and these were run throughout the development process to verify the concurrent implementation provides the same results and the baseline implementation.

Besides using the test framework for testing performance and correctness of the implementations, other quality attributes can be useful in determining the concurrency strategy. Quality attributes are part of the non functional requirements to an application or framework. Therefore, the three concurrency features: Parallel collections, futures, and actors were evaluated based on other quality attributes as well. To use the results of the tests and evaluations, it was necessary to analyze them, to be able to generalize on the usage of concurrency in the COE and choose the best concurrency strategies.

The following section describes principles, that the testing should adhere to in order to get usable

results. Next, the test framework that enables automatic testing is described in section 4.3 and the tests are described in section 4.4. Because the testing strategy was not ideal, it is reflected on in section 4.5. Afterwards, the additional quality attributes and a rating of these are described in section 4.6. The results of the automatic tests and the evaluation of quality attributes are described in section 4.7 and finally, section 4.8 presents the analysis of the results.

4.2. Principles

Certain principles should be adhered to when testing to get usable results, and these principles are presented in this section. The approach to testing in this thesis project focuses on getting usable results without implementing or using extensive test tools and avoiding large test specifications, as testing is not the primary goal. However, to ensure usable results in the context of this thesis some principles should be adhered to:

Test environment: A test consisting of multiple simulations should performed on the same hardware with approximately the same processes running during the test. The reason for stating ”approximately the same processes“ is, that the tests were run in a Windows environment, where it is not possible to completely control the running processes from the Operating System. All processes irrelevant to the execution of tests should be disabled during the tests.

Test functions: To limit inconsistencies in the processes running between simulations, each test should be implemented as a single test function. This means, that a test performing simulations on the baseline and the three concurrent implementations should be implemented in one test function to avoid undesirable interaction required to start other tests. To further ensure usable results the COE application should be restarted for every simulation.

Correct simulation results: The baseline is considered to be an oracle and it is assumed that it calculates the ”correct“ simulation results. It should be verified that the concurrent implementations calculate the same simulation results as the baseline implementation.

Automation: The tests should be automated so they are easy to replicate and less prone to manual errors. This will also make them usable in the future development of the COE.

Modifying Functional Mock-up Units (FMUs): If an FMU is modified it should be carried out in such a way that it has minimal impact on the execution time of a simulation.

4.3. Test Framework

This section introduces the test framework, see figure 4.1, that was developed to enable testing of the different COE implementations. The purpose of the test framework was to support performance testing and verification of consistency between the results of a baseline simulation and the results of any of the concurrent simulations. Because the baseline and each of the concurrent implementations were developed separately, it was necessary to provide support for launching different executables. As mentioned in section 3.2.1, the COE application runs as a web server and

it was therefore also necessary to provide support for the HTTP requests: initialize, simulate, and results. These combined provided the basis for testing simulations on COE executables. The code for the test framework is shown in appendix E along with an example of a test. The classes of the test framework in figure 4.1 are described below:

IOThreadHandler: The purpose of this class is to print the output from the COE application. In general, this class encapsulates a thread that prints the output of a process.

CoeLauncher: The purpose of this class is to launch an executable with an implementation of the COE, possibly with arguments, and return the process to be used by the IOThreadHandler described above. It general, this class launches an exectuable Jar file with the given arguments and returns the process.

ServerRequester: The purpose of this class is to provide support for the necessary HTTP requests: initialize, simulate, and results, that are required to perform a simulation and retrieve the results. This is done with POST and GET requests, described in section 3.2.1, tailored specifically for this COE.

SimulationResults: The purpose of this class is to store execution times, simulation results, and whether the simulation results matches the baseline results.

TestRunner: The purpose of this class is to orchestrate an entire simulation by using the classes mentioned above. It is possible to pass the result of a baseline simulation, that will be compared to the result of the simulation being performed. As the name `RunTestsForJar` suggests, this function performs testing using a single executable; however each argument passed will lead to a separate simulation, where the argument is passed to the COE application upon launch. Thus passing two arguments will lead to two simulations, one being performed using the COE application launched with the first argument and another using the COE application launched with the second argument.

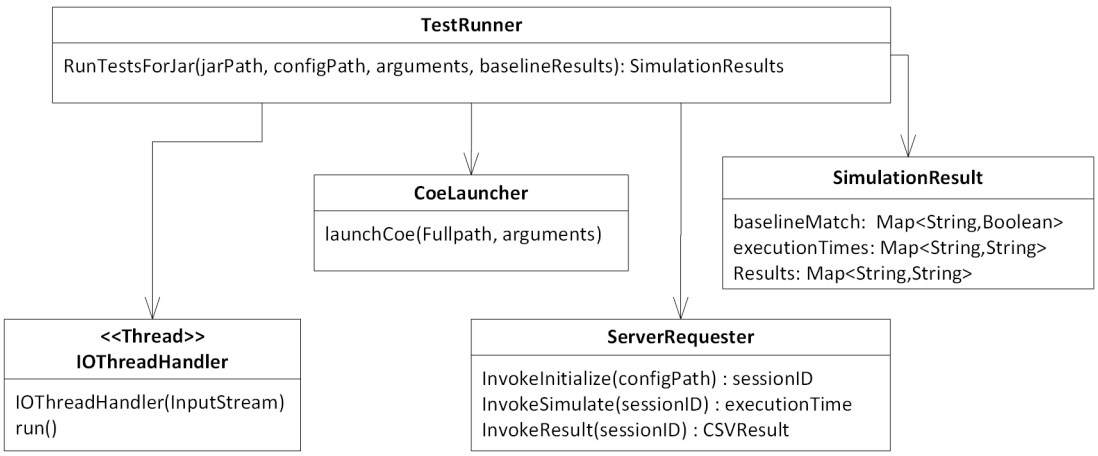


Figure 4.1: Class diagram of the test framework

To aid in the development of test scenarios, a shell script was created to copy an FMUs, modify the configuration file of the copied FMU, and modify the COE configuration file to make use of the additional FMUs. The shell script can be seen in appendix F, and to better understand the script the structure of an FMU package is described in section 2.4.1. In the case where the initial simulation setup looks as in figure 4.2 and the script is run to add two additional integrate FMUs chained to the original integrate FMU, it will result in the setup in listing 4.3.

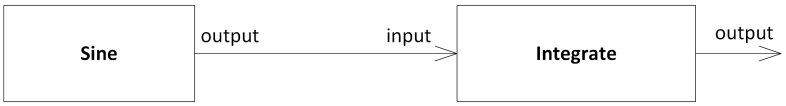


Figure 4.2: Example of two connected FMUs.

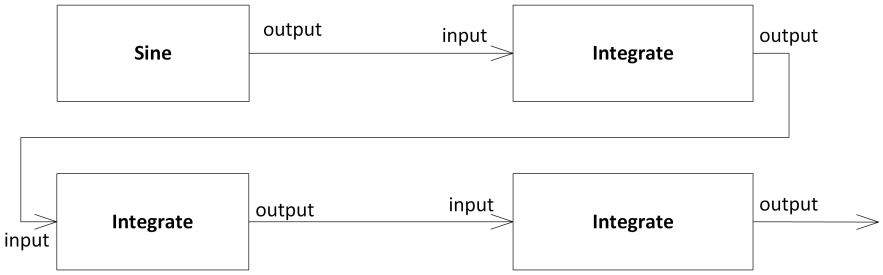


Figure 4.3: Example of four connected and chained FMUs.

4.4. Performance Tests

This section presents the tests used for evaluating the performance of the baseline and the different concurrency implementations along with validating, that the concurrent implementations calculate the same simulation results as the baseline implementation. The tests described in this section makes use of the framework and shell script described in section 4.3 and will adhere to the principles described in section 4.2. More specifically, each test will be implemented in a single test function, each simulation will compare the results from the baseline simulation with the results of the concurrent simulations, and all processes irrelevant to running the test will be disabled.

The basic test flow of a simulation can be seen in figure 4.4. This shows how a given simulation is performed five times using the baseline implementation of the COE and then averaging the execution times. Next, the given simulation is run five times for each concurrent implementation and then averaging the execution time. Lastly, the simulation results are compared and the execution times are stored. The COE application is restarted between every simulation, as suggested in section 4.2.

First section 4.4.1 tests the implementations using non-modified heating, ventilation, and air conditioning (HVAC) FMUs to illustrate the performance of the implementations in a real simulation. Afterwards, the test described in section 4.4.2 uses modified FMUs in order to evaluate the performance in a more competent manner. This is done to get a rough estimate as to when the concurrent implementations perform a simulation faster than the baseline implementation and provide a basis for further testing.

4.4.1 Heating, Ventilation, and Air Conditioning Tests

The HVAC tests will be used to investigate the performance of the baseline implementation and the concurrent implementations in a simulation using non-modified FMUs. The simulation uses five FMUs, one controller FMU and four Fan Coil Unit (FCU) FMUs, as shown in figure 4.5. The test follows the test flow shown in figure 4.4. The FMUs are described below:

Controller: The controller FMU can control the water flow into the FCU FMU via the valve actuator and the state of the damper via the damper actuator. It has an input to receive the

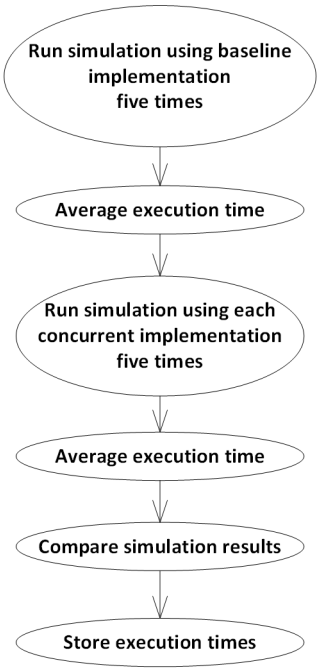


Figure 4.4: Basic test flow of a simulation using baseline and concurrent implementations.

temperature of the return air from the FCU FMUs.

FCU: The FCU FMU receives input on how to set the state of the valve that controls the water flow and the state of the damper. It outputs the temperate of the return air to the controller FMU.

Three tests using these FMUs will be performed to see how concurrency behaves in different scenarios. Each of the simulations in the tests has an end time of 1000 seconds and uses one controller FMU and four FCU FMUs. The tests are described below:

HVAC Test1 (1 Con, 4 FCU, 0.1 step): The purpose of this test is too see, how the implementations perform with a small step size and with real FMUs.

Step size: 0.1 seconds.

Implementations used: Baseline described in section 3.2, implementation using parallel collections described in section 3.3.2, implementation using futures described in section 3.3.3, and implementation using actors described in section 3.3.4.

HVAC Test2 (1 Con, 4 FCU, 20 step): The purpose of this test is too see, how the implementations perform with a large step size and with real FMUs.

Step size: 20 seconds.

Implementations used: Baseline, implementation using parallel collections, implementation using futures, and implementation using actors.

HVAC Test3 (1 Con, 4 FCU, 0.1 step, MultipleActorInvocations): The purpose of this test is to see how the implementations perform before and after the refactoring.

Step size: 0.1 seconds.

Implementations used: Multiple actor invocations with the argument `ALL` described in section 3.3.5, multiple actor invocations with the argument `DOSTEP`, and implementation using actors.

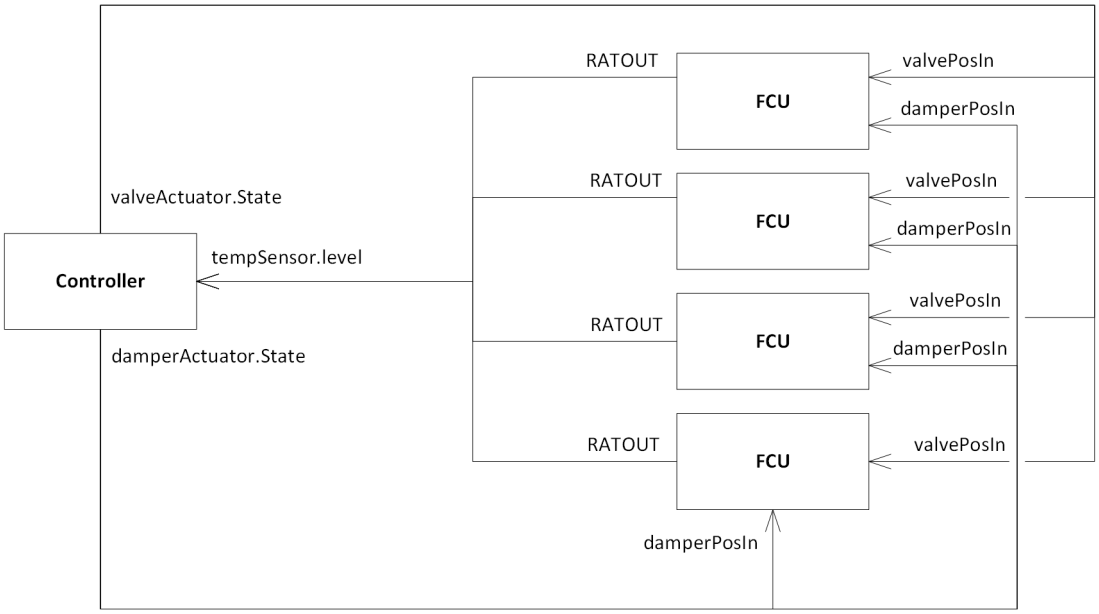


Figure 4.5: HVAC simulation with one controller FMU and four FCU FMUs.

4.4.2 Sine Integrate Wait Tests

These tests will be used to investigate the performance of the baseline implementation and the concurrent implementations by using modified FMUs, which will be elaborated on the section 4.4.3. The FMUs used in these tests are the following:

Sine The sine FMU generates a sine wave with amplitude and angular frequency of the wave set to one as shown in figure 4.6. It is possible to modify the amplitude and the angular frequency in the configuration file.

Integrate The integrate FMU integrates the sine values, as shown in figure 4.6.

Modified Integrate The modified integrate FMU integrates the sine values, as shown in figure 4.6. It differs from the integrate FMU above, as it has been modified to read a value from a file representing wait time, and wait with the duration of the wait time in the `doStep` function. The reading of a value from a file is performed in a function, that is invoked during the initialization of an FMU and not during the simulation, and therefore it does not affect the execution time. Figure 4.7 shows how the different wait times are implemented in the tests that uses modified FMU.

Several tests using these FMUs will be performed to see how concurrency behaves in different scenarios. Each of the simulations in the tests has an end time of 100 seconds and a step size of 0.1 seconds. The tests are described below:

SI Test1 (1 Sine, 1 Integrate): The purpose of this test is to see how the implementations perform with one increasingly demanding FMU and one non-demanding FMU.

FMUs: 1 sine FMU and 1 modified integrate FMU as illustrated in figure 4.2.

Wait time: The wait time values will start at zero milliseconds and stop at one millisecond with 0.1 millisecond intervals.

Implementations used: Baseline described in section 3.2, implementation using parallel collections described in section 3.3.2, implementation using futures described in section 3.3.3, and implementation using actors described in section 3.3.4.

SI Test2 (1 Sine, 5 Mod-Integrate): The purpose of this test is to see how the implementations perform with five increasingly demanding FMUs and one non-demanding FMU.

FMUs: 1 sine FMU and 5 modified integrate FMUs chained as illustrated in figure 4.3.

Wait time: The wait time values will start at zero milliseconds and stop at one millisecond with 0.1 millisecond intervals.

Implementations used: Baseline, implementation using parallel collections, implementation using futures, and implementation using actors.

SI Test3 (1 Sine, 100 Integrate): The purpose of this test is to see how the implementations perform with many non-demanding FMUs.

FMUs: 1 sine FMU and 100 integrate FMUs chained as illustrated in figure 4.3.

Implementations used: Baseline, implementation using parallel collections, implementation using futures, and implementation using actors.

SI Test4 (1 Sine, 100 Integrate, MultipleActorInvocations): The purpose of this test is to see how the implementations perform before and after the refactoring.

FMUs: 1 sine FMU and 100 integrate FMUs chained as illustrated in figure 4.3.

Implementations used: Multiple actor invocations with the argument `ALL` described in section 3.3.5, multiple actor invocations with the argument `DOSTEP`, and implementation using actors.

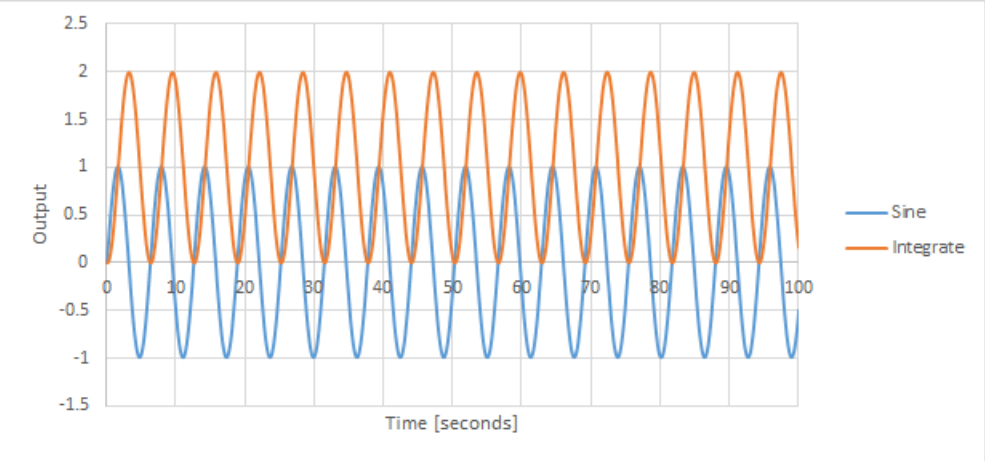


Figure 4.6: Sine FMU and integrate FMU simulation results.

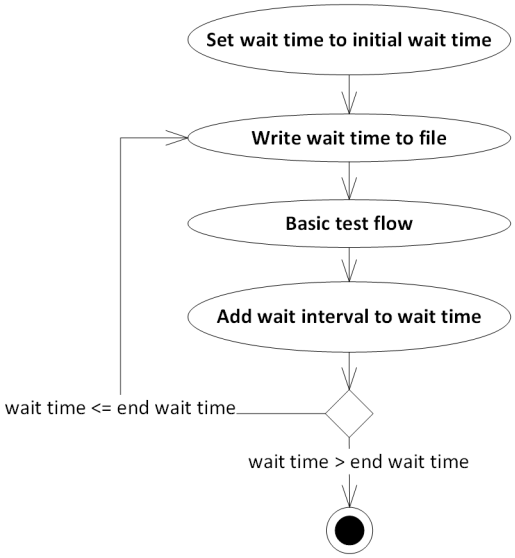


Figure 4.7: Test flow of a simulation using baseline and concurrent implementations with wait FMUs, see section 4.4.2.

4.4.3 Modified FMU

This section describes how an FMU was modified to wait for a certain amount of time. A time to wait is read from a file in the function `fmiInstantiate`, which is invoked during the loading of FMUs in the initialization phase of a simulation described in section 3.2.1. This wait time is used in the `doStep` function of the FMU using the function `usleep2` shown in figure 4.1. This makes use of `QueryPerformanceCounter` which is based on a hardware counter and recommended by Microsoft to use when high-resolution time stamps are required with microsecond precision [Microsoft, 2015]. The function call in line three in listing 4.1 gets the current number of ticks performed since Windows was started and the function call on line four gets the number of ticks per second. In the do-while loop a new number of ticks performed is retrieved, and these three pieces of information can be used to measure the elapsed time. The frequency is divided by 1000000 to get the frequency in microseconds, which is then multiplied by the wait time read from a file and compared to the elapsed time. The argument to the function therefore has to be specified in microseconds. The function uses busy-waiting, but it is acceptable in this case.

C

```
1 void uSleep2(int waitTime) {
2   __int64 time1 = 0, time2 = 0, frequency = 0;
3   QueryPerformanceCounter((LARGE_INTEGER *) &time1);
4   QueryPerformanceFrequency((LARGE_INTEGER *)&frequency);
5   do {
6     QueryPerformanceCounter((LARGE_INTEGER *) &time2);
7   }while((time2-time1) < ((frequency/1000000) * waitTime));
8 }
```

Listing 4.1: Wait function in FMU

The function `usleep` was used initially, as it is part of the library `unistd`, which was already included in the FMU. `usleep` suspends the execution of a thread for at least the number of microseconds passed to the function [die.net, 2015]. However initial testing with one sine FMU and one modified FMU showed, that this did not have any effect until 1000 microseconds as it can be seen in table 4.1.

Sleep	Baseline	Future	Par	Actor
0.0	64	142	217	112
100.0	61	124	211	130
200.0	59	126	214	117
300.0	68	135	221	117
400.0	60	119	208	108
500.0	62	122	215	116
600.0	82	158	284	125
700.0	73	135	233	111
800.0	60	139	208	104
900.0	59	117	210	114
1000.0	1926	1973	2105	1947

Table 4.1: Results in milliseconds from using `usleep`

4.5. Reflection on Testing

The test framework supports automated testing of simulations in terms of launching the COE, running simulations, reporting simulation results, and reporting simulation execution times. The test framework performs the testing by invoking the COE as it would be invoked in a real use scenario and therefore have minimal impact on the COE execution. Therefore, the test framework provides a foundation for testing the ongoing development of the COE.

Even though the tests are automated, they have to be started manually, and they are run on a local computer with a minimum of other processes running. Thereby the state of the computer is confined to running only the test related processes and the necessary Operating System processes, which blocks the use of the computer for other activities. This makes it a time-consuming process to perform testing, which is not desirable. The results are reported as csv and JSON files and every test creates at least two data files. Therefore, to take advantage of the data created by executing a test several times involves manually opening and putting the data together, because new files are generated in every test. This is also time-consuming and it makes reasoning about the results increasingly difficult. An alternative is to setup a server to run all the tests on a regular basis and store the results in a database. This would make it a single time-consuming process to set up the server and therefore not a continuing process of blocking the local computer, and it would make it easier to generate a large data set, which would make reasoning easier and provide better results. This was not established because of the time constraints of this thesis project.

It was difficult to find FMUs for testing, and it was difficult to find tools that support exporting FMUs that perform long-lasting computations. This made it necessary to create the modified FMU as described in section 4.4.3 to get an idea of the performance of the implementations using concurrency. The modified FMU performs busy-waiting in the `doStep` function, which represents

an idealized situation. It is idealized because the additional computations does not depend on the inputs, step size, state of the FMU etc. The HVAC tests uses real FMUs, which illustrates that using concurrency can increase the performance of the COE without altering the FMUs. However, these tests are also idealized along with the Sine Integrate tests described in section 4.4.2 because of the way that correctness is established. The correctness depends on whether the baseline computes the same result for the same simulation as mentioned in section 4.2. This makes it possible to alter the step size, add additional FMUs to a simulation and still get the correct results. However, these results might not be correct, if the original configuration of the simulation was to be run with e.g. a lower or higher step size, or it was expected that the stability of the FMUs was evaluated during the simulation and so forth.

4.6. Additional Quality Attributes

In order to choose the best suited concurrency strategy for a COE it is necessary to evaluate more quality attributes than performance, which was described in section 4.4. A quality attribute can be used as a nonfunctional requirement to an application, or in the context of this thesis an evaluation parameter to be used in evaluating the different concurrent implementations and the different concurrent strategies: parallel collections, futures and actors introduced in section 2.8. The additional quality attributes that will be used the evaluate the strategies in relation to the COE are described below:

Composability: The purpose of the composability attribute is to evaluate the possibility of combining elements in different ways to satisfy the necessary requirements. As described in section 2.3 FMUs can have different step sizes and rollbacks can be necessary. This can lead to complicated scenarioes, where composability becomes important.

Simplicity: The purpose of the simplicity attribute is to evaluate the implementation effort required to implement a given concurrency strategy, and how easy it is to understand the implementation. It should be clear what the purpose of an implementation is, and the implementation should be simple, as stated by the KISS (Keep It Simple, Stupid) principle. Simplicity of the concurrent implementations will be evaluated by considering whether it is clear why it is necessary to invoke certain functions, what they do, and how they can be used. As the COE should be able to run simulations using many different FMUs and configurations along with being part of a tool chain as described in section 2.6 it can be expected to change. If it is simple, this change will be easier to implement.

Configurability: The purpose of the configurability attribute is to evaluate, how much the frameworks can be configured to suite different requirements without having to implement new functionality. As it is impossible to know which kind of simulations the COE will be used for, one solution does not necessarily fit all. However, if a solution is configurable, it might suit more needs and thereby preventing the need for additional implementation, or it can be configured to perform better.

Scalability: The purpose of the scalability attribute is to evaluate, how the frameworks support increased work load. The evaluation will look at how the frameworks support splitting the workload over additional hardware, e.g. networked computers. As the COE is running as a web server, described in section 3.2.1, scalability is an important attribute.

Documentation: The purpose of the documentation attribute is to evaluate, how well the frameworks are documented. As the concurrency strategy can have a significant effect on the execution time and the COE can be expected to run on different systems it is important, that the frameworks behave as expected, and that the usage of the frameworks is optimized. To achieve this, it is necessary with good documentation.

The frameworks will be given a rating of -, +, or ++ for each attribute along with a description of why the given rating was achieved. The ratings are described below in table 4.2.

Property	**Rating**		
	-	**+**	**++**
Composability	No composability	Limited composability	Easy composability
Simplicity	Extensive initialization and complicated usage	Some initialization and moderate usage	Basic initialization and easy usage
Configurability	Not configurable	Moderately configurable	Very configurable
Scalability	No functionality for scaling	Some scaling functionality	Extensive scaling functionality
Documentation	Limited documentation	Basic documentation	Well documented and many resources
Performance	Slowest	In the middle	Fastest

Table 4.2: Rating of quality attributes. Performance is relative to the three implementations.

4.7. Results

This section contains the results of the tests described in section 4.4 and the quality attributes described in section 4.6. Section 4.7.1 contains the results for the HVAC test and section 4.7.2 contains the results for the Sine Integrate Wait Tests.

The results of the performance tests are presented in tables. The unit is milliseconds, and the meaning of the table heads are described below:

Baseline: This is the execution time of the baseline implementation.

Future: This is the execution time of the implementation using futures.

Par: This is the execution time of the implementation using parallel collections.

Actor: This is the execution time of the implementation using actors.

ActorAll: This is the execution time of the implementation multiple Actor Invocation with the argument `ALL`.

ActorDoStep: This is the execution time of the implementation multiple Actor Invocation with the argument `DOSTEP`.

(NAME)Diff This is the execution time of the leftmost implementation in the table subtracted with the execution time of (NAME) in the table.

Wait This is the wait time.

4.7.1 HVAC Tests

Several tests were performed using the HVAC controller FMU and the HVAC FCU FMU to see how the implementations perform with different wait times and with a different number of FMUs. The test cases are described in section 4.4.1 and the results are shown below.

HVAC Test1 (1 Con, 4 FCU, 0.1 step): The results from this test can be seen in table 4.3. Ranked in order of lowest execution time: Implementation using futures, implementation using actors, baseline implementation, and implementation using parallel collections.

HVAC Test2 (1 Con, 4 FCU, 20 step): The results from this test can be seen in table 4.4. Ranked in order of lowest execution time: Implementation using futures, implementation using actors, implementation using parallel collections, and baseline implementation.

HVAC Test3 (1 Con, 4 FCU, 0.1 step, MultipleActorInvocations): The results from this test can be seen in table 4.5. Ranked in order of lowest execution time: Implementation using actors, the implementation multiple actor invocation with the argument `DOSTEP`, and then the implementation multiple actor invocation with the argument `ALL`.

Baseline	Future	Par	Actor	FutureDiff	ParDiff	ActorDiff
31256	29822	31980	30919	1434	-724	337

Table 4.3: Results in milliseconds from HVAC Test1 (1 Con, 4 FCU, 0.1 step) described in section 4.7.1.

Baseline	Future	Par	Actor	FutureDiff	ParDiff	ActorDIff
26591	23951	24284	24263	2640	2307	2328

Table 4.4: Results in milliseconds from HVAC Test2 (1 Con, 4 FCU, 20 step) described in section 4.7.1.

Actor	ActorAll	ActorDoStep	ActorAllDiff	ActorDoStepDiff
30407	31709	30968	-1302	-561

Table 4.5: Results in milliseconds from HVAC Test3 (1 Con, 4 FCU, 0.1 step, MultipleActorInvocations) described in section 4.7.1.

4.7.2 Sine Integrate Wait Tests

Several tests were performed using the sine FMU, the integrate FMU, and the modified integrate FMU to see how the implementations perform with different wait times and with a different number of FMUs. The test cases are described in section 4.4.2 and the results are shown below.

SI Test1 (1 Sine, 1 Integrate): The results from this test can be seen in table 4.6. Ranked in order of lowest execution time: Baseline implementation, implementation using futures, implementation using actors, and then the implementation using parallel collections.

SI Test2 (1 Sine, 5 Mod-Integrate): The results from this test can be seen in table 4.7. The baseline was faster than any of the concurrent implementations with wait time set to zero. With wait times above zero and ranked in order of lowest execution time: Implementation using futures, implementation using `actors`, implementation using parallel collections, and baseline implementation.

SI Test3 (1 Sine, 100 Integrate): The results from this test can be seen in table 4.8. Ranked in order of lowest execution time: Implementation using futures, baseline implementation, implementation using actors, and then implementation using parallel collections.

SI Test4 (1 Sine, 100 Integrate, MultipleActorInvocations): The results from this test can be seen in table 4.9. Ranked in order of lowest execution time: Implementation using actors, the implementation multiple actor invocation with the argument `DOSTEP`, and then the implementation multiple actor invocation with the argument `ALL`.

Wait	Baseline	Future	Par	Actor	FutureDiff	ParDiff	ActorDiff
0	195	330	656	374	-135	-461	-179
100	1036	1178	1609	1249	-142	-573	-213
200	1892	2035	2546	2109	-143	-654	-217
300	2748	2901	3417	2966	-153	-669	-218
400	3608	3773	4289	3837	-165	-681	-229
500	4468	4635	5161	4715	-167	-693	-247
600	5325	5502	6034	5587	-177	-709	-262
700	6185	6359	6920	6453	-174	-735	-268
800	7043	7223	7792	7308	-180	-749	-265
900	7900	8082	8663	8173	-182	-763	-273
1000	8758	8938	9545	9032	-180	-787	-274

Table 4.6: Results in milliseconds from SI Test1 (1 Sine, 1 Integrate) described in section 4.7.2.

Wait	Baseline	Future	Par	Actor	FutureDiff	ParDiff	ActorDiff
0	355	434	834	622	-79	-479	-267
100	4642	1252	1976	1357	3390	2666	3285
200	8925	2085	2460	2164	6840	6465	6761
300	13280	2981	3408	3116	10299	9872	10164
400	17554	3838	4241	3909	13716	13313	13645
500	21904	4679	5042	4746	17225	16862	17158
600	26195	5510	6212	5616	20685	19983	20579
700	30448	6405	6820	6489	24043	23628	23959
800	34760	7267	7643	7376	27493	27117	27384
900	39251	8174	8921	8313	31077	30330	30938
1000	43356	8970	9348	9184	34386	34008	34172

Table 4.7: Results in milliseconds from SI Test2 (1 Sine, 5 Mod-Integrate) described in section 4.7.2.

Baseline	Future	Par	Actor	FutureDiff	ParDiff	ActorDiff
1464	1432	1967	1857	32	-503	-393

Table 4.8: Results in milliseconds from SI Test3 (1 Sine, 100 Integrate) described in section 4.7.2.

Actor	ActorAll	ActorDoStep	ActorAllDiff	ActorDoStepDiff
2099	4302	2463	-2203	-364

Table 4.9: Results in milliseconds from SI Test4 (1 Sine, 100 Integrate, MultipleActorInvocations) described in section 4.7.2.

4.7.3 Evaluation of Quality Attributes

The evaluation of the frameworks based on the quality attributes described in section 4.6 is presented in appendix G. Additionally, the results of the evaluations are shown in a spider chart in figure 4.8. The evaluation of the performance quality attribute is based on the results from section 4.7.1 and 4.7.2, which means it is relative to the three implementations.

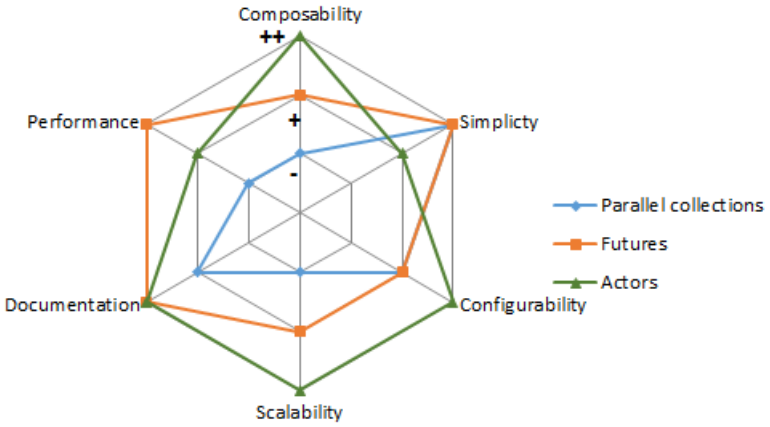


Figure 4.8: Evaluation of quality attributes. The rating is described in table 4.2.

4.8. Analysis of the Results

To use the results for improving further development, it is necessary to draw some conclusions. Therefore, this section describes some conclusions that can be drawn from the results in section 4.7. The structure of this section consists of first presenting the conclusions based on the results. Afterwards it will be discussed, what these conclusions can be used for.

The refactored concurrent implementation is faster than the non-refactored concurrent implementation: The performance of the implementation using actors based on the baseline described in section 3.3.5 was compared to the performance of the implementation using actors based on the refactored implementation described in section 3.3.4 and the results can be seen in section 4.7, table 4.5 and table 4.9. The results show, that the actor implementation based on the refactored implementation called ”Actor“ in the tables is the fastest. As the implementation using futures described in section 3.3.3 is faster in all of the tests than the implementation using actors it is reasonable to conclude, that this implementation is faster as well. Furthermore, the data for ”Actor“ and ”ActorDoStep“ supports, that performing more computations separated in threads that are spawned anyhow is faster, than performing fewer computations in threads and doing the remaining computations sequentially. This supports that performing the refactoring to limit the number of concurrent invocations and the synchronizations was the correct step to take in order to lower the execution time.

Invoking several functions in one concurrent invocation with one synchronization is faster, than invoking several functions in separated concurrent invocations with multiple synchronizations: This conclusion is based on the same comparison as the conclusion above, namely the results in section 4.7, table 4.5 and table 4.9. The results of ”Actor“ and “ActorAll” show, that invoking several functions in one concurrent invocation with one synchronization is faster, than invoking several functions in separated concurrent invocations with multiple synchronizations.

Overhead of invoking functions concurrently is higher than what is gained by invoking the same functions sequentially: This conclusion is based on the same comparison as the conclusion above, namely the results in section 4.7, table 4.5 and table 4.9. The data for ”ActorDoStep“ compared to “ActorAll” shows, that invoking the `doStep` function concurrently and running the rest of the simulation steps sequentially is faster, than invoking several functions in separated concurrent invocations and invoking fewer functions sequentially. This test shows, that the price of invoking functions concurrently might be higher than what is gained by invoking the same functions sequentially.

Executing simulations sequentially can be faster than executing them concurrently: Performing simulations sequentially can be faster than running them concurrently, which is backed by the results for SI Test1, SI Test2, SI Test3, and HVAC Test1. More noticeably is it, that even introducing an additional millisecond of computation in the `doStep` function of an FMU as tested in SI Test1 does not make the simulation faster using concurrency than without concurrency. Because the baseline is faster than the concurrent implementations in several tests with two or more FMUs, the hypothesis is considered to be refuted.

Executing simulations concurrently can be faster than executing them sequentially: SI Test2 shows, that if several FMUs perform an additional millisecond of computations in the `doStep` function, then it is faster to run the simulation using concurrency. This contradicts the previous conclusion and thereby indicates, that deciding whether or not to run a simulation concurrently cannot be decided by looking at whether or not a single FMU perform long-lasting computations. It is necessary to look at more than one FMU being used in a given simulation. Besides looking

at how long-lasting the computations of a given FMU are, the data from SI Test3 shows, that the amount of FMUs in a simulation plays a role. The simulation in SI Test3 performs faster without concurrency, as it was shown in SI Test1, but the sheer number of FMUs made one of the concurrent solutions perform the simulation faster. Even though the hypothesis is refuted because of the conclusion above, this shows, that concurrency can still increase the performance of a simulation.

From here we conclude, that it is necessary to allow for different simulation strategies to achieve the fastest simulation. These strategies should support running simulations sequentially, concurrently, or a mix. For example, if an FMU that performs long-lasting computations is to be simulated with three FMUs that performs fast computations, then it could be optimal to run this simulation using two threads as in figure 4.9. This figure shows an example, where the Master Algorithm (MA) splits the simulation into two concurrent processes: One process simulates a single FMU that performs long-lasting computations and the other process sequentially simulates three FMUs that performs fast computations. To implement this, it requires a level of composability. To properly determine the optimal strategies it is necessary to perform more tests with different FMUs and different scenarios, which will be further elaborated on in section 6.5.

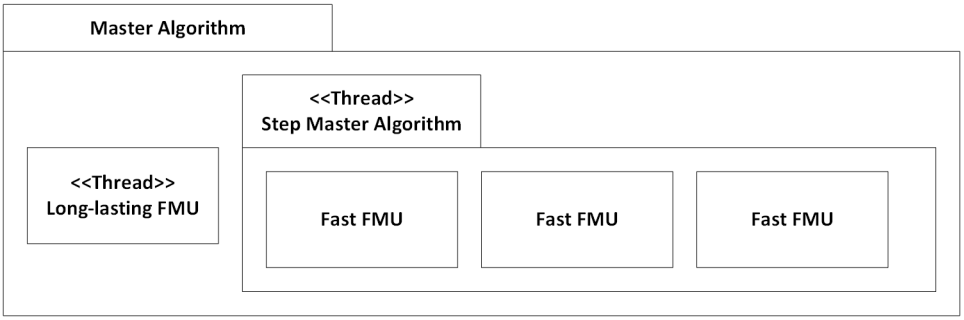


Figure 4.9: Master algorithm simulating four FMUs using an additional step master algorithm, that performs a single step.

Allowing for different strategies also involves computing the strategy to use. A way of assisting the choice of strategy is including a measure of how long-lasting the computations performed by an FMU are in the FMU configuration file, which may vary from where the FMU is in the simulation as well. However, this file is standardized, and therefore it might not be possible. Furthermore, choosing the best strategy might not depend on individual FMUs, but all the FMUs used in a given simulation, and therefore this solution would still include processing. Instead of storing this metadata in the FMU configuration file, it can be stored in the simulation configuration file. This would allow for targeting specific simulations and the combination of FMUs used in a given simulation. The metadata can be established either by defining it in the simulation configuration file or implementing functionality in the COE to store the strategy used. This would require the COE to perform computations that processes the execution time of the FMUs used in a simulation and calculate the best strategy. This could be carried out by adapting or switching strategy between simulation steps, and choosing the fastest. As simulations are expected to be reused several times with adaptions to the FMUs deployed, it might be worth spending resources on computing and storing the best strategy once, so it can be reused. Furthermore, as the COE is part of a tool chain dedicated to developing Cyber-Physical Systems (CPSs) as described in section 2.6, the metadata can be shared across multiple tools and possibly improve the performance of these tools as well.

The different features used for performing the functions concurrently are not equally fast, with the implementation using parallel collections being the slowest and the implementation using fu-

tures the fastest. As the implementation using parallel collections does not offer any additional advantages over the implementation using futures as shown in section 4.7.3, it can be ruled out for this purpose, unless future tests show otherwise. However, the implementation using actors offers advantages in terms of scalability, configurability and composability. As the COE is implemented as a web server, scalability can be an important attribute to service many clients or perform distributed simulations in the future development of the COE. Configurability can help to adapt the COE for different setups, and as the COE is expected to be run by several companies, it should be configurable. Composability can prove to be an important attribute in implementing different strategies, as it was detailed in the paragraph above.

Chapter 5

Guidelines

This chapter presents guidelines on adding functionality that takes advantage of concurrency to improve performance for an existing application, and therefore helps to achieve goal 3. It is based on the work performed in this thesis, but it is applicable in other contexts as well. Because it is based on the work performed in this thesis, it is a natural continuation of the previous chapters, and it is part of the concluding remarks and future work in chapter 6. However, to make these guidelines more accessible this chapter is also roughly self-contained, though it will contain references to relevant parts of this thesis.

5.1. Introduction

The Intel co-founder Gordon E. Moore has made some interesting observations and predictions on the evolution of processors. One of these observations and predictions is famously known as "Moore's law": *"The number of transistors incorporated in a chip will approximately double every 24 months."* [as cited in Intel Corporation, 2015][1]. Roughly this translates to: The speed of processors will double every two years. Besides observing this fact, it was also a prediction that has proven to be a guideline for the semiconductor industry. However, it has started to slow down, and Intel is having trouble keeping up with this "law" [Reisinger, 2014]. So, as the increase in processor speeds has begun slowing down, the manufacturers have turned to multiple processing cores known as *multicore* [Geer, 2005; Creeger, 2005]. This requires software developers to take advantage of concurrency in order to make use of the additional cores.

This chapter describes some guidelines on implementing functionality that takes advantage of concurrency in an existing application. They do not cover distributed systems, language specific features, or threading strategies as such. First, section 5.2, presents a motivation for creating these guidelines and what they hopefully achieves. Afterwards, section 5.3 to 5.5 describes the guidelines for adding functionality that uses concurrency in an existing application. These are split up in three parts: 1) Preparation/Prelimary steps, 2) implementation, and 3) evaluation. Figure 5.1 presents an overview of the guidelines and will be explained in the sections below.

[1] Originally, Moore stated, that it would double every year [Moore, 1965], but it was revised in 1975 [as mentioned in Mollick, 2006].

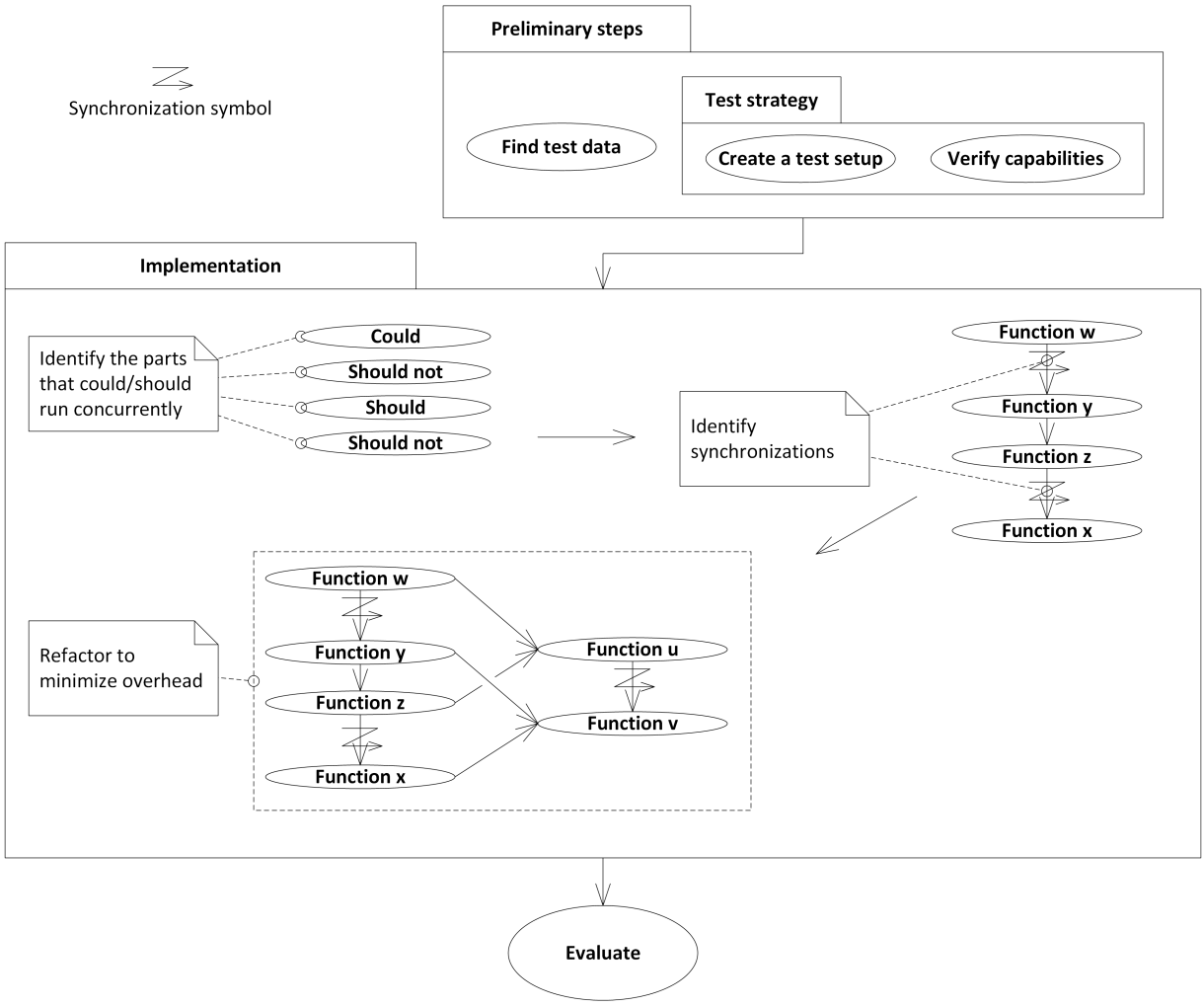


Figure 5.1: Guidelines on implementing concurrency.

5.2. Motivation for Guidelines

An adage known by ”Wirth’s law“ goes: *”Software is getting slower more rapidly than hardware becomes faster“* [Wirth, 1995].[2] To compensate for this slowing, Wirth states: *”The way to streamline software lies in disciplined methodologies and a return to the essientials.“* [Wirth, 1995]. The motivation for creating these guidelines is to present a few steps that can be used to take advantage of concurrency to increase the performance of software in terms of execution time. It is the hope, that these guidelines can be turned into a methodology in the future.

Concurrency can take advantage of multiple cores by performing two or more computations concurrently instead of sequentially. When implementing functionality that uses concurrency in an application, it increases the complexity, as it becomes necessary to consider the challenges involved. *Race conditions* is one of these challenges (race conditions are described in section 2.7.2), and it can happen when two or more threads are accessing the same resource, and their execution order depends on the scheduling of threads [Helmbold and McDowell, 1996]. Race conditions can lead to nondeterminism, which is undesirable, as it is more difficult to reason about. To ensure

[2]Wirth attributes this to a different saying by Reiser [Reiser, 1991]

determinism one can use synchronization mechanisms, e.g. *semaphores*, but these comes with other challenges. For example *deadlocks*, which happens when two or more threads are waiting for a resource, that is held by another. This makes concurrency seem difficult, but it does not necessarily have to be.

There are concurrency features, that abstracts the necessary synchronization, so it is not necessary for the developer to deal with these challenges. The Scala example in listing 5.1 takes advantage of concurrency via the function `par` to download three images from a web server. This concurrency feature is called parallel collections (see section 2.8.3 for a more thorough description), and what it does is splitting the list `imageUrls` into three concurrent tasks, and when these have finished downloading an image each, these images are combined back into a list. This is abstracted from the developer and therefore it is not necessary to handle synchronization explicitly.

Scala

```
1 val imagesUrls = List("http://example.dk/img1.png","http://
    ↪ example.dk/img2.png","http://example.dk/img3.png")
2 val images = imagesUrls.par.map(imageUrl => downloadImage(
    ↪ imageUrl))
```

Listing 5.1: Download images from a url concurrently

The example in listing 5.1 is small and conceptually simple; but if the implementation is structured in a fashion that is described in these guidelines, then the same principles can be applied in other contexts and to a larger amount of functionality. The hope is, that these guidelines will help make developers more inclined to try using concurrency and take advantage of the possibilities it offers.

5.3. Preparing for Concurrency

Before implementing functionality that uses concurrency in an application, it can be useful to address some challenges to avoid complications later in the implementation. One of these challenges is investigating whether the implementation is a ”one size fits all“ implementation in the sense, that it will only use one concurrency strategy. A strategy can determine which parts of the functionality is executed concurrently, and which are not. In the case of ”one size fits all“ it is expected, that the domain is well-known and therefore it is evident, that the chosen strategy results in the desired benefits. In the other case, the domain is unknown or the benefits that is achieved from the choice of strategy is unclear. Because of this, it can be difficult to determine which strategy to implement, or whether to implement several strategies. For this reason, it can be useful to conduct tests and in order to test it is necessary to have the following: Test data that covers the different use cases that the application expect to fulfill and a test setup that makes testing possible.

It might not be possible to find test data covering all the use cases at an early stage. However, this does not mean, that this step should be ignored. In some cases it is possible to create test data where the amount of computations is parameterized, i.e. the modified Functional Mock-up Unit (FMU) described in section 4.4.3, which can make it possible to rule out some strategies or indicate, that additional testing is necessary to determine the right strategies. This involves early testing, where the application has not yet been implemented to support concurrency to the full extent and therefore this should not include too much effort. The implementation described in

section 3.3.5 is an example of this. Collecting test data should be a priority throughout the project, as it will enable testing of the application throughout the project, which can prove valuable; especially when implementing several strategies.

Implementing several strategies is not always a simple task, and different strategies can be better at particular cases. Because it can be difficult to reason about which strategy is the best for different cases, *regression testing* is important in finding the optimal strategy for different cases. Regression testing is testing for regressions, where *"a regression is a feature that used to work and now doesn't"* [Osherove, 2009]. Without going into the definition a feature, it can be tests which verify, that changes have not introduced performance degradations or wrong results. In case of multiple strategies, it can be necessary to test and compare the results. However, if the implementation is not carried out with a focus on testability, it can prove difficult to do this. The initial testing of the Co-Simulation Orchestration Engine (COE) involved several conditionals as shown in listing 3.8, and this made it difficult to extend and configure. Therefore, deciding on a test setup early can ensure, that the implementation is carried out with testability in mind, and this makes it easier to test and identify the optimal strategies for different cases. If concurrency usage is being added to an existing implementation similar to what was done in this thesis project, then it is also important to clearly determine the capabilities of the existing implementation and create tests, that verify this behavior.

Creating a test setup before starting the implementation resembles the idea behind Test-Driven Development (TDD) [Beck, 2002], where a failing test is created before implementing the functionality that makes it pass. However, the test setup does not provide clear results on whether a given strategy is the best strategy, because it may depend on other quality attributes, that are difficult to measure objectively. Therefore, the test setup can be used to evaluate the implementation based on some measurable quality attributes, while others must be addressed in a different manner.

5.4. Process of Implementing Use of Concurrency

The previous section focused on what could be performed as preliminary steps to applying concurrency such as finding test data and deciding on a test setup. This section describes the process of developing an application to take advantage of concurrency. Several of the steps that were performed in the process of making the COE take advantage of concurrency (described in chapter 3) can be applied to implementing the use of concurrency in other systems. The following describes guidelines that can be used to implement the usage of concurrency in other applications, and examples from this thesis project will be used to demonstrate some of the principles.

The first step is to identify which parts of the application that could run concurrently. In this thesis project, this step revealed several parts as described in section 3.3.1: Setting inputs, serializing state, invoking `doStep`, performing rollbacks, and retrieving output variables. It is not always straightforward to identify these parts, and it is important to make a distinction between parts that *could* run concurrently and parts shat *should* run concurrently. For example, the task of setting inputs in the FMUs consists of assignment operations, and the overhead in terms of time of using concurrency for this task would be larger than the time gained.

Once the parts than can run concurrently have been identified, it is necessary to consider the synchronization involved in making these parts run concurrently. Synchronization requires waiting for the threads to finish their computations, and is considered part of the overhead by using concurrency. In section 5.2 a small example was mentioned, where synchronization could be abstracted

away. This is still true, as the low-level details are abstracted away, but it is still necessary to mark the synchronization spots in the implementation, and the overhead still exists. The COE was implemented in such a way, that it was necessary to synchronize several times in a simulation step as shown in figure 3.4, and the implementation using multiple actor invocations described in section 3.3.5 performs several synchronizations.

Assume that the parts that can run concurrently and the synchronization related to running them concurrently has now been identified. This provides a basis for determining a refactoring of the application as to minimize the overhead of using concurrency. The refactoring should focus on maximizing the amount of computations for every thread and minimizing the synchronizations. Figure 5.2 shows this process, where the identification of concurrent part and synchronization before the refactoring has been performed on the left hand side, and after the refactoring on the right hand side. The left hand side is a typical sequential implementation, which follows the pattern: Prepare, Act, Process Results (PAPR). These steps happen just as the application is expected to behave: Prepare for a task, execute the task, process the results, prepare for the next task etc. However, each of the concurrent operations comes with an overhead in initializing the threads and each of the synchronizations comes with an overhead of waiting for the threads to finish their computations. Because of this, it is necessary to change process of repeating PAPRs into as few coherent concurrent functions and synchronizations as possible, which is what happens when going from left to right in the figure. Instead of preparing for a single task, prepare for as many concurrent tasks as possible. This limit is defined by, how many tasks can be done independently in different threads before a synchronization is necessary. When the preparation for multiple concurrent functions has been performed, these can be run with only a single thread initialization for each *set of coherent functions*. A set of coherent functions are functions related to a particular task, for example downloading and scaling an image is a set of coherent functions (an example using such functions is given below). The processing of results requires all the results, and therefore a synchronization is necessary, before performing the last step. In the implementation of the COE it was possible to perform only one synchronization; however, this is not always the case.

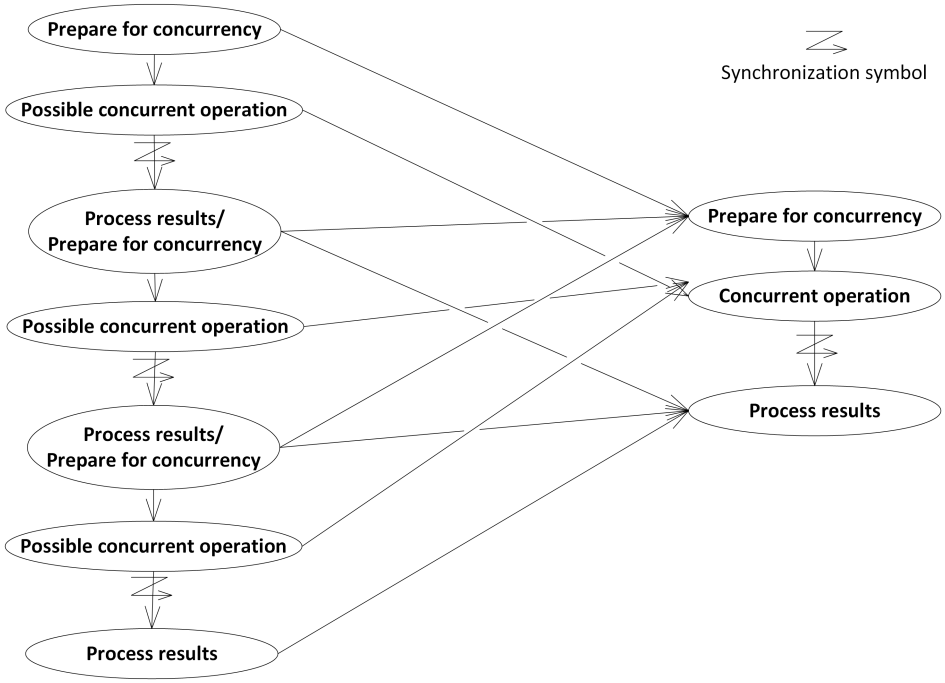


Figure 5.2: Reduction of separated concurrent invocations and synchronizations.

If the image example from listing 5.1 is extended to both download and scale the images, then the process mentioned above can be applied. This is shown in figure 5.3, where the left hand side and the right hand side performs the following:

Left hand side: First the URLs of the images are set, then all the images are downloaded, and this is verified. Then the scaling properties are set, all the images are scaled, and the are passed to the UI for viewing or something else.

Right hand side: First set the URLs and the scaling properties of the images. Then perform the following set of functions for each image: download and scale the image. These functions are performed in a single concurrent invocation for each image. Lastly, verify the images and pass them to the UI or something else.

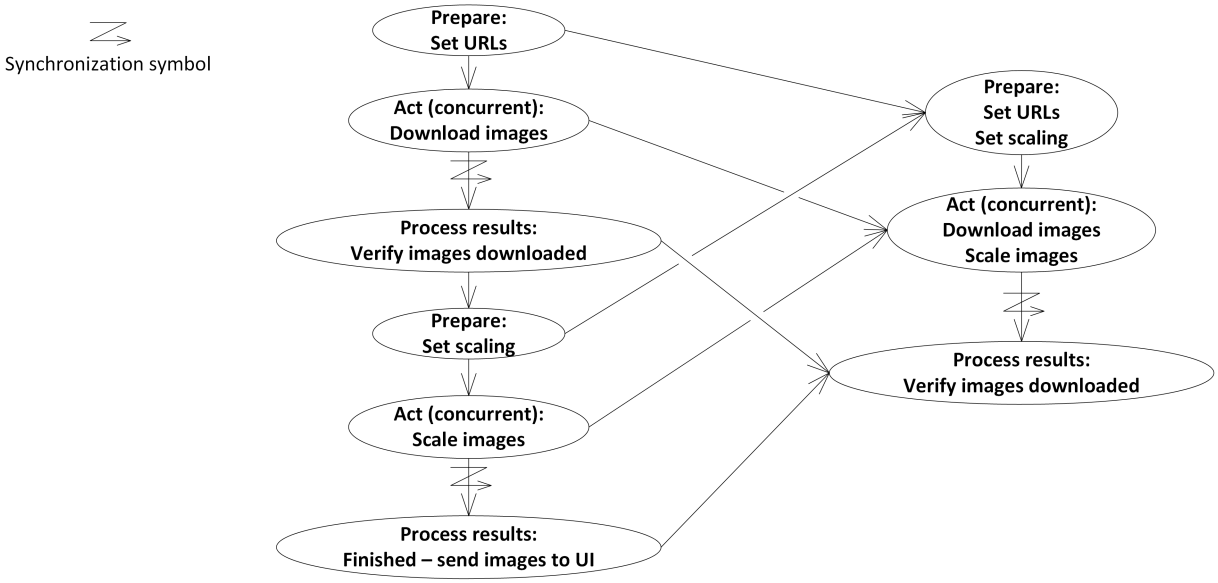


Figure 5.3: Reduction of separated concurrent invocations and synchronizations for the image example.

This refactoring successfully reduces two separate concurrent invocations to one and two synchronizations to one. However, there is a trade-off: Consider the process of downloading three images: `img1`, `img2`, and `img3`. Say `img1` fails to download, but `img2` and `img3` succeeds. In the left hand side example this would be detected immediately after the download. In the right hand side example, `img2` and `img3` would still be scaled, before the failure to download `img1` is detected. This can lead to unnecessary computations and should be considered when performing the refactoring.[3] It is worth noting, that the concurrent functions should be *referential transparent*, which can be roughly described as without side-effects (a more thorough explanation was given in section 2.7.4). When executing code in parallel that has side-effects, it may affect the code in several threads in an unpredictable fashion and thus lead to undesired behavior.

[3]Several programming languages support terminating threads in a case like this. However, that increases the complexity and is not considered applicable in general.

5.5. Evaluating an Implementation

Once the implementation is finished, the test setup described in section 5.3 can be used to evaluate it. It is necessary to verify the *correctness* of the different strategies meaning, that they calculate the correct results; i.e. in a COE setting, the results must be semantically consistent. The performance testing provides a measure of how fast the strategies perform calculations and can be used to fine-tune the strategies or detect cases where the application uses a less optimal strategy.

Performance and correctness are only part of the quality attributes, that can be used to evaluate the implementation. Other attributes such as scalability and composability can be necessary to provide additional features, but it might result in sacrificing performance. For example, the trade-off mentioned above can also be used to verify the implementation. Therefore, it can be necessary to choose which quality attributes a given application should exhibit, and how important these are in relation to each other. For example, if a given strategy fulfills a use case in an optimal fashion according to performance, and this use case covers the vast majority of users. However, for a small number of users it is not the optimal strategy, but it still covers their use case. Implementing another strategy that is optimal for these users is considered to include an increase in complexity and degradation of maintainability of the entire application. Should the additional strategy be implemented? Evaluation of an implementation should be used to combine the implementation with the quality attributes, that the application is expected to exhibit.

As applications are subject to change, the evaluations must also be subject to change. When more features are implemented and more test data becomes available, the strategies may have to be adjusted or new ones have to be developed. Because of this, testing is an ongoing task throughout the entire development process, as this provides additional data for reasoning about the implementation. The tests should track the quality attributes over time, so tendencies can be clarified and ensure, that the development of the application targets the selected quality attributes. This should be set up in an automated fashion, for example on a build server.

Chapter 6

Concluding Remarks and Future Work

This chapter concludes this thesis by presenting the main findings, and it is therefore based on all of the previous chapters. Besides presenting the main findings, this chapter also evaluates on the achievement of the goals and hypothesis stated in chapter 1. As this thesis only scratches the surface of concurrency and the implementation of the Co-Simulation Orchestration Engine (COE) was not finished at the end of this thesis project, this chapter also outlines possible future work.

6.1. Introduction

Reflection on work that has been carried out can be used to improve the process of other projects and learn from mistakes. Therefore, it is discussed in the next section which parts of this thesis project that could have been performed in a better fashion, but also new skills achieved by the author of this thesis. Next, section 6.3 presents the conclusion containing the main findings of this thesis. Besides presenting the main findings, this thesis also had four goals set, and these are evaluated on in section 6.4. Afterwards, future work for both the development of the COE and the guidelines are outlined in section 6.5. Finally, section 6.6 presents some final remarks on the work performed in this thesis.

6.2. Discussion

In order to learn from the process in this thesis project, it is necessary to reflect on how it has been carried out. An important part of this is identifying parts, that has been challenging. The identification of these parts makes it possible to avoid making the same mistakes in a future project, or take measures to alleviate them. Furthermore, it is important to identify skills that are applicable in general and should be continuously improved.

Previously, in section 4.5 the test framework was reflected upon. Next, section 6.2.1 describes part of the approach that proved to be challenging and could have been carried out differently. Finally, section 6.2.2 describes transferable skills, which are skills that are applicable in general.

6.2.1 Thesis Project Approach

The approach described in section 1.5 has proven to be a challenge, because the COE was at an early stage in development and not tested thoroughly when this MSc thesis started. Bugs have been found after establishing the baseline and that has impacted the development. For example, the Functional Mock-up Units (FMUs) were not unloaded correctly after a simulation, and therefore the web server had to be relaunched for every simulation. This bug would have been found before establishing a baseline if the testing of the COE had been more comprehensive. Because of the missing tests it is also difficult to determine which features the baseline implementation supports, and thereby which features the concurrent implementations should support. Therefore, a vital part of establishing a baseline should be to clearly identify what is supported and have tests to verify this. This put additional requirements on establishing a baseline and potentially delays the investigation of concurrency, but ensures the implementation is correct in terms of supported functionality and possibly avoids having to do workarounds due to bugs.

Besides the challenges regarding establishing a baseline described above, the chosen approach has also made the continued work on the COE unnecessary complicated. This is caused by the fact, that the COE has also been developed externally in parallel to this thesis project, and thus it will be difficult to merge the refactored implementation and the external development of the COE, because the developments have taken different branches. This means, that depending on the success of this thesis project, it will most likely be a matter of reimplementing the refactoring on the external development instead of merging the two developments. A way of accomplishing a better reuse of this thesis project would be to merge the external development and the refactored implementation as shown in figure 6.1 as soon as the refactored implementation was finished. In this way, the external development could have continued based on the refactored implementation, and the implementations in this thesis project would still not be impacted by the external development.

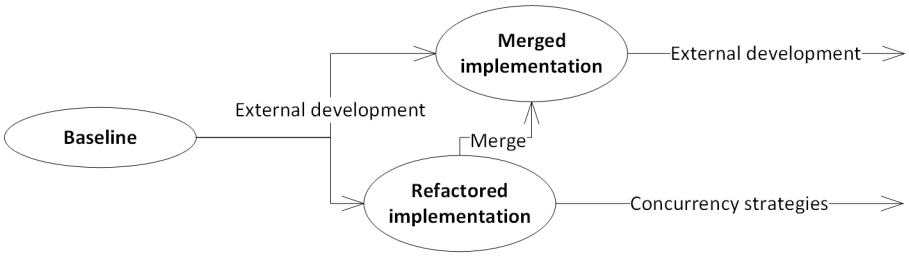


Figure 6.1: Server requests for performing a simulation and retrieving the results.

6.2.2 Transferable Skills

Acquiring new information
Acquiring information was a challenge, because there is a lot of literature. It is therefore not feasible to expect to read every article on a given subject, as there is simply too much. This is quite different from taking a course, where the literature is usually provided by the teachers, and it requires a different strategy. One of the first steps should be to get an overview of the available literature by reading abstracts and use this for categorization. Thereby some resources may be considered off topic, and other resources will prove to be on topic. A resource that is particularly useful in relation to the given topic often contains references to other useful resources. The trans-

ferable skill is therefore to improve in acquiring and filtering new information and thereby reuse earlier research to perform better research.

Source criticism/Information evaluation
Source criticism is important and not always straightforward. A source can be credible and considered an expert within an area, but still provide information aiming at a specific purpose, that might not be ideal. Therefore it can be necessary to consider information in the functional view *"... which states that a source is not in itself good or bad, but just more or less fruitful or relevant in relation to a given question"* [Hjørland, 2012].

During this study I encountered a case that required additional attention. Martin Odersky is the developer of Scala and as such considered an expert within the Scala community. He also founded the company Typesafe together with the creator of the framework Akka, which is being promoted as the default framework for actors in Scala [Jovanovic and Haller, 2015]. It is important to keep this in mind, for example when Martin Odersky states: *"That said, currently by far the most popular approach to concurrency is Scala's actor library."* [Sommers, 2008] or appears in Typesafe related content. The transferable skill is to be critical when reading literature and identify possible underlying motivations.

Collaboration
Collaboration is a large part of doing research and the author of this thesis has been surprised with how helpful and open to discussion other researchers are. A part of collaboration is working together achieving the desired results, another part is also showing that the collaboration is beneficial. As the INTO-CPS project has several industrial and academic partners, collaboration is required. During this thesis project a partner asked for help on making a quick demonstration of the COE for a meeting occurring two days later, and the only thing available for running an entire simulation was a shell script. However, as the partner was familiar with Java, the test framework developed during this thesis project was used to set up a configurable test, that performed an entire simulation. It is important to create understandable content during the research to make the progress clear to collaborators. The transferable skill is to make sure, that research is available to others while being conducted as well as after.

Conveying information
Sir Isaac Newton once wrote: *"If I have seen farther, it is by standing upon the shoulders of giants."* [as cited in Brewster, 2009]. Conveying information is a fundamental part of researching, as it lays the foundation for other research to build upon. To properly convey information it is important to be precise and define all terms, that can lead to doubt. A way of conveying a definition of a term is citing a resource that describes it, thereby the reader can read the cited source if in doubt, otherwise a description of the term is necessary, which may prove difficult. Besides precisely defining terms, it is also important to clearly show how different concepts relate to each other and how the given research is applicable in general. Structuring this thesis and creating the flow that leads the reader from one subject to the next has been a challenge to the author, and should be focused on in the future. Conveying information is also closely related to collaboration, as effective conveying of information can improve the collaboration. The transferable skill is to become better at conveying information, thereby making the research more usable.

6.3. Conclusion

The thesis project is based on an application called the COE which performs Co-Simulation without using concurrency. Functionality that takes advantage of concurrency was implemented using Scala in three different versions of the COE, each using different concurrency features. The concurrency features were the following: Parallel collections, futures, and actors. In order to evaluate the different implementations a test framework was developed. The framework supports launching the COE, invoking the COE to perform a simulation, and retrieving the results and execution time of a simulation.

The test framework was used to evaluate the implementations in terms of correctness and execution time. For this purpose, different FMUs and test scenarios were set up. The implementation using futures was faster in terms of execution time than the implementation using actors, which again was faster than the implementation using parallel collections. However, the original sequential implementation was faster in some scenarios involving two or more FMUs. Because the sequential implementation was faster than the concurrent implementations in these cases, the hypothesis has been proved to not hold for all simulations with two or more FMUs and is therefore refuted. It is necessary to investigate whether implementing multiple strategies that combines sequential and concurrent processing can optimize the performance of the COE. Other quality attributes than performance may also be important to the further development of the COE, and therefore the implementations were evaluated based on the following quality attributes: Composability, simplicity, configurability, scalability, and documentation. The results are shown in figure 6.2.

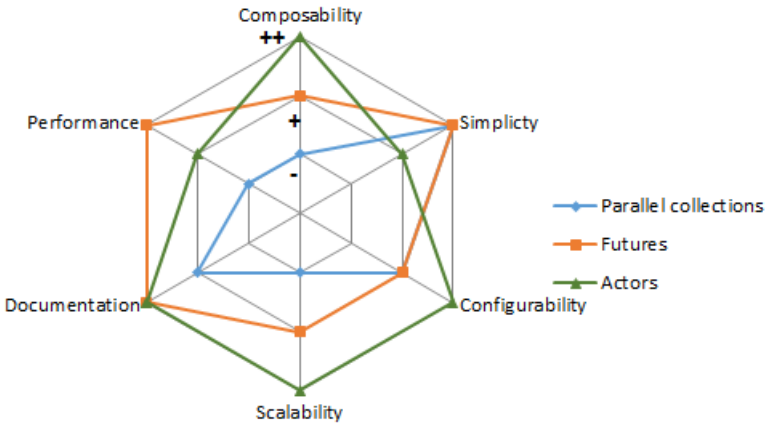


Figure 6.2: Evaluation of the chosen quality attributes for the COE. The rating is described in table 4.2.

The steps used for implementing functionality that uses concurrency in this thesis project can be applied in other contexts, and therefore the process was generalized to the following steps:

Preliminary tasks: Concurrency may not increase the performance under all conditions. To determine the optimal strategy for when to use concurrency or how to use concurrency it is necessary to test the application under different conditions. To ensure the application is testable, a test strategy should be decided on early, and test data should be found.

Identify concurrent parts: Some parts in the application are better suited to execute concur-

rently than other parts. It is necessary to identify which parts that *should* execute concurrently, which parts that *could* run concurrently, and which parts that *should not* run concurrently.

Identify synchronizations: Executing parts of the application concurrently often requires coordination during the execution of the application. Therefore, the different parts have to synchronize and this might involve waiting until the concurrent parts have finished computing. The synchronization required for coordinating the tasks that could/should run concurrently, which was identified in the previous step, should be identified.

Implement concurrency: Based on the identification performed in the previous two steps, the implementation of functionality that uses concurrency should try to achieve the following: Minimize the invocation of concurrent functions, maximize the work load in every concurrent function, minimize the number of synchronizations. To achieve the best performance, it can be necessary to implement different strategies that determines whether to use concurrent processing, sequential processing, or a mix and which feature to use for concurrency.

Evaluate the implementation: Evaluate and test the implementation based on selected quality attributes for the application. Based on the results from evaluating and testing the implementation, the strategies may need to be adjusted.

Ongoing evaluation: When the application is used in a new way, this should result in new test data and thereby possibly an adjustment of the strategies. Automatic regression testing should be used, to ensure the application targets the selected quality attributes over time.

6.4. Evaluation on the Achievement of the Goals

Besides investigating and testing the hypothesis, this thesis project also had four goals that should be accomplished. This section revisits these goals and evaluates whether they have been accomplished or not.

6.4.1 Revisting the goals

In order to evaluate the achievement of this thesis project, four goals were set. The goals were presented in chapter 1, and shown again below:

Goal 1: Understand the concepts of Cyber-Physical Systems (CPSs) and Co-Simulation. Additionally, learn how Co-Simulation can be performed using the Functional Mock-up Interface (FMI) standard.

Goal 2: Learn the basics of the challenges that implementing the usage of concurrency presents, and how the functional paradigm addresses these.

Goal 3: Learn and use the programming language Scala for the development of the COE application with different concurrency features. A part of this goal is to be able to generalize on implementing functionality that uses concurrency in an application.

Goal 4: Improve personal skills in general for conducting research by identifying transferable skills that may be reusable in the future.

6.4.2 Evaluation

The evaluation of the goals set for this thesis project is described in this section. Each goal will be evaluated by briefly explaining why the goal is considered to be achieved, and afterwards which parts of this thesis are related to the goal.

Goal 1: This goal is *successfully achieved*. The concepts of CPSs and Co-Simulation were studied and grasped, along with understanding a relation between these concepts. Furthermore, FMI has been studied along with algorithms describing how to use it in practice. These algorithms have laid the foundation for the implementation of an application that performs Co-Simulation using FMI.

Literature on the concepts of CPSs and Co-Simulation has been studied and the concepts were described in section 2.2 and section 2.3. The description of FMI consists of some background in section 2.4 and actual algorithms for performing Co-Simulation using FMI in section 2.5. Moreover, the concepts were related to each other in section 1.1 and the implementations described in chapter 3 have performed Co-Simulation using FMI.

Goal 2: This goal is *successfully achieved*. The reason for using concurrency and the main challenges in using concurrency has been studied. Furthermore, the solutions to these challenges offers new challenges, which were studied as well. The functional paradigm addresses these challenges, and Scala has been used exemplify the concepts.

Section 2.7 describes why it is important to use concurrency, but also the challenges inherent in using it. Afterwards, section 2.8 presented how the functional paradigm addresses these challenges.

Goal 3: This goal is *successfully achieved*. The COE application is developed in Scala, so the refactoring has performed in Scala. Furthermore, three different concurrency features: parallel collections, futures, and actors were used in the development of the COE. The process of refactoring, implementing, and evaluating the implementation of the concurrency features in the COE were generalized to a set of guidelines, than can be used in other projects.

The basics of the concurrency features used in the implementation was described in section 2.8, and they have been used for the implementations described in chapter 3. The guidelines based on the implementation in chapter 3 and evaluation in chapter 4 was described in chapter 5.

Goal 4: This goal is *successfully achieved*. This thesis is the result of conducting research for the first time by the author, and therefore several transferable skills have been improved. These are considered to be general skills and therefore applicable within many areas. Because this thesis does not deal with transferable skills as such, the evaluation of this goal deviates from the evaluation of the other goals above. These skills were described in section 6.2.

6.5. Future Work

To achieve more knowledge within the subject of performing Co-Simulations, more studies and tests are required. Furthermore, it is interesting to gain more knowledge on implementing functionality that uses concurrency in general, which also can be applied to this subject. Below, some possibilities are listed, which can be used to continue the work from this thesis.

Implement features: The implementation of the COE is not finished, as there are still features that need to be supported such as extrapolation and interpolation of simulation step results. These features need to be implemented.

Testability: The COE currently supports reporting the execution time of an entire simulation without initialization and reporting of results. The initialization and reporting of results will be part of any simulation, and therefore these should also be tested for performance. Additionally, the COE should offer better granularity for performance measurements. A better granularity makes it possible to examine the performance of different parts of the COE, and this can be used to target the development effort and find bottlenecks.

Testing: To gain more knowledge of when to use sequential or concurrent processing it is necessary to gather more real test data. The test data should be diverse in the sense that the FMUs should have different computation times, amount of inputs/outputs and step sizes, but also diverse in the COE configuration of a simulation. Different configurations can lead to different scenarios, where rollbacks, extrapolation and interpolation of the step results are needed. To better take advantage of the additional tests a test server should be set up to automatically run performance tests. The performance tests should be run on every commit, so it is clear whether the commit increases of decreases the performance. Besides focusing on individual commits, it should also be possible to view the results of the performance tests over time. This will help to illustrate the tendency of the implementation and provide a foundation for discussing whether the architecture of the COE is correct.

Strategy: This thesis has shown, that sequential processing is faster than concurrent processing in some cases and vice versa. Therefore, it is necessary to investigate, whether an increase of performance is achievable by enabling sequential, concurrent, and mixed processing. Mixed processing is where some FMUs are grouped together and run sequentially inside the group, but runs concurrent to other groups of FMUs or single FMUs as shown in figure 4.9.

Initialization and adaption: If the investigation of choosing strategies results in implementing multiple strategies, then it should be investigated how to properly configure the COE, so it chooses the right strategy for optimal performance of a given simulation. One way to do this is including metadata in the COE configuration file, which should describe the performance of the different FMUs and choice of strategy. Another way is to let the COE adapt between running simulation steps based on the performance of FMUs. Once the COE has found the optimal strategy, this can be serialized to the COE configuration file for the given simulation.

Guidelines The guidelines provide basic advice on implementing functionality that uses concurrency in an existing application. It could be interesting to use these guidelines on different case studies to gain more knowledge of cases where they can be applied, and cases where they cannot be applied. This would outline cases where the guidelines are lacking, and this can be used for extending the guidelines. Furthermore, existing research on the topic should be studied, as it may contribute to subjects already covered by the guidelines or introduce subjects, that should be covered. The hope is, that these guidelines are extended and improved, thereby gradually becoming a methodology.

6.6. Final Remarks

The hypothesis was, that implementing concurrency increases the performance of the COE. This has been proven to be false in some cases and true in others. This does not mean, that the work done in this thesis is useless, but that a different approach to increasing the performance of the COE has to be chosen. The approach must combine sequential and concurrent processing to achieve optimal performance.

Besides testing the hypothesis, this thesis project has also successfully achieved four goals that were set up. The achievement of these goals mean, that the author has studied the fields of CPSs, Co-Simulation using FMI, concurrency, the functional paradigm, and Scala. Furthermore, the work carried out in this thesis project has improved the research skills of the author.

Implementing functionality that uses concurrency in the COE also gave rise to some guidelines on the process. This expands the relevance of this thesis from concerning the process of adding concurrency usage to a specific application to a more general applicable process, that can be applied to a multitude of applications.

To conclude this thesis, it is the hope that the results are used in the further development of the COE, and that the guidelines are used in other projects and improved. Additionally it is the hope, that Co-Simulation becomes widespread in the development of model-based CPSs and improves both the development process and the end result of such systems.

Bibliography

[Agha and Kim, 1999] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762198000678 [cited at p. 29, 30, 79]

[Armstrong, 2007] J. Armstrong, "A history of erlang," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 6–1–6–26. [Online]. Available: http://doi.acm.org.ez.statsbiblioteket.dk:2048/10.1145/1238844.1238850 [cited at p. 29, 79]

[Ashcroft, 1975] E. Ashcroft, "Proving assertions about parallel programs," *Journal of Computer and System Sciences*, vol. 10, no. 1, pp. 110 – 135, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000075800183 [cited at p. 24, 79]

[Avizienis et al., 2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, Jan 2004. [cited at p. 12, 79]

[Baker and Hewitt, 1977] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/800228.806932 [cited at p. 26, 79]

[Bastian et al., 2011] J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "P.: Master for co-simulation using fmi," in *8th International Modelica Conference*, 2011. [cited at p. 2, 11, 14, 19, 79]

[Beck, 2002] Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. [cited at p. 66, 79]

[Bird and Wadler, 1988] R. Bird and P. Wadler, *An Introduction to Functional Programming*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988. [cited at p. 24, 79]

[Box and Draper, 1987] G. E. P. Box and N. R. Draper, *Empirical Model-building and Response Surfaces*. Wiley, 1987. [cited at p. 13, 79]

[Brewster, 2009] D. Brewster, "Chapter vi (diplomatic version)," http://www.newtonproject.sussex.ac.uk/view/texts/diplomatic/OTHE00101, September 2009, (Visited on 12/16/2015). [cited at p. 73, 79]

[Broman et al., 2013] D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, Sept 2013, pp. 1–12. [cited at p. 15, 18, 80, 91]

[Chiusano and Bjarnason, 2014] P. Chiusano and R. Bjarnason, *Functional Programming in Scala*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2014. [cited at p. 25, 80]

[Creeger, 2005] M. Creeger, "Multicore cpus for the masses," *Queue*, vol. 3, no. 7, pp. 64–ff, Sep. 2005. [Online]. Available: http://doi.acm.org.ez.statsbiblioteket.dk:2048/10.1145/1095408.1095423 [cited at p. 2, 12, 22, 23, 63, 80]

[die.net, 2015] die.net, "usleep(3) - linux man page," http://linux.die.net/man/3/usleep, Dec 2015, (Visited on 12/07/2015). [cited at p. 53, 80]

[Dijkstra, 1965b] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, pp. 569–, Sep. 1965. [Online]. Available: http://doi.acm.org/10.1145/365559.365617 [cited at p. 23, 80]

[Dijkstra, 1965a] ——, "Cooperating sequential processes, technical report ewd-123," Tech. Rep., 1965. [cited at p. 22, 23, 24, 80]

[Dijkstra, 1971] E. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971. [Online]. Available: http://dx.doi.org/10.1007/BF00289519 [cited at p. 22, 80]

[Elsheikh et al., 2013] A. Elsheikh, M. Awais, E. Widl, and P. Palensky, "Modelica-enabled rapid prototyping of cyber-physical energy systems via the functional mockup interface," in *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on*, May 2013, pp. 1–6. [cited at p. 16, 80]

[EPFL, 2015] EPFL, "Implicit parameters - scala documentation," http://docs.scala-lang.org/tutorials/tour/implicit-parameters.html, 2015, (Visited on 12/10/2015). [cited at p. 80, 110]

[FMI development group, 2014a] FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf [cited at p. 11, 16, 80, 87, 88]

[FMI development group, 2014b] ——, "Fmi [start]," https://fmi-standard.org/, 2014, accessed May 28 2015. [cited at p. 15, 80]

[Friedman and Wise, 1976] D. P. Friedman and D. S. Wise, "The impact of applicative programming on multiprocessing," in *Proceedings of the 1976 International Conference on Parallel Processing*. Long Beach, CA, USA: IEEE, 1976, pp. 263–272, IEEE Catalog Number: 76CH1127-0C. [cited at p. 26, 80]

[Geer, 2005] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005. [cited at p. 2, 12, 22, 23, 63, 80]

[Gill, 1958] S. Gill, "Parallel programming," *The Computer Journal*, vol. 1, no. 1, pp. 2–10, 1958. [Online]. Available: http://comjnl.oxfordjournals.org/content/1/1/2.abstract [cited at p. 22, 23, 80]

[Haller and Odersky, 2009] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202 – 220, 2009, distributed Computing Techniques. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397508006695 [cited at p. 29, 30, 81]

[Haller et al.] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," http://docs.scala-lang.org/overviews/core/futures.html, (Visited on 12/10/2015). [cited at p. 81, 111]

[Hansen, 1973] P. B. Hansen, *Operating System Principles*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1973. [cited at p. 24, 81]

[Helmbold and McDowell, 1996] D. Helmbold and C. McDowell, "A taxonomy of race conditions," *Journal of Parallel and Distributed Computing*, vol. 33, no. 2, pp. 159 – 164, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731596900349 [cited at p. 23, 64, 81]

[Hewitt et al., 1973] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org.ez.statsbiblioteket.dk:2048/citation.cfm?id=1624775.1624804 [cited at p. 29, 81]

[Hibbard, 1977] P. Hibbard, "Parallel processing facilities," in *New directions in algorithmic languages, 1976*, S. A. Schuman and I. de recherche d'informatique et d'automatique (France)), Eds. 2, 1976, Saint-Pierre-de-Chartreuse, IsÃ¨re: IRIA, 1977, pp. 1–7. [Online]. Available: http://opac.inria.fr/record=b1075190 [cited at p. 26, 81]

[Hjørland, 2012] B. Hjørland, "Methods for evaluating information sources: An annotated catalogue," *Journal of Information Science*, vol. 38, no. 3, pp. 258–268, 6 2012. [cited at p. 73, 81]

[Hoare, 1985] T. Hoare, *Communication Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985. [cited at p. 31, 81]

[Intel Corporation, 2015] Intel Corporation, "Moore's law and intel innovation," http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html, December 2015, (Visited on 12/18/2015). [cited at p. 63, 81]

[ITEA Office Association, 2015] ITEA Office Association, "Itea 3 Â· project Â· 07006 modelisar," https://itea3.org/project/modelisar.html, December 2015, (Visited on 12/06/2015). [cited at p. 15, 81]

[Jovanovic and Haller, 2015] V. Jovanovic and P. Haller, "The scala actors migration guide - scala documentation," http://docs.scala-lang.org/overviews/core/actors-migration-guide.html, 2015, (Visited on 10/23/2015). [cited at p. 73, 81]

[Katz and Weise, 1990] M. Katz and D. Weise, "Continuing into the future: On the interaction of futures and first-class continuations," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: ACM, 1990, pp. 176–184. [Online]. Available: http://doi.acm.org/10.1145/91556.91628 [cited at p. 27, 81]

[Larsen, 2015] P. G. Larsen, "into-cps.au.dk," http://into-cps.au.dk/, 2015, (Visited on 10/13/2015). [cited at p. iii, 21, 81]

[Lea, 2000] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 Conference on Java Grande*, ser. JAVA '00. New York, NY, USA: ACM, 2000, pp. 36–43. [Online]. Available: http://doi.acm.org.ez.statsbiblioteket.dk:2048/10.1145/337449.337465 [cited at p. 31, 82]

[Lee, 2008] E. A. Lee, "Cyber physical systems: Design challenges," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8, Jan 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html [cited at p. 1, 5, 12, 82]

[Lee, 2010] ——, "CPS foundations," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 737–742. [cited at p. 2, 12, 13, 82]

[Microsoft, 2015] Microsoft, "Acquiring high-resolution time stamps (windows)," https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx, 2015, (Visited on 12/07/2015). [cited at p. 52, 82]

[Mollick, 2006] E. Mollick, "Establishing moore's law," *IEEE Annals of the History of Computing*, vol. 28, no. 3, pp. 62–75, 2006. [cited at p. 63, 82]

[Moore, 1965] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965. [cited at p. 63, 82]

[Netzer and Miller, 1992] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130616.130623 [cited at p. 23, 82]

[Newcombe et al., 2014] C. Newcombe, T. Rath, F. zhang, B. Munteanu, marc Brooker, and M. Deardeuff, "Use of formal methods at amazon web services," http://research..com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf, 2014, accessed May 27 2015. [cited at p. 13, 82]

[NIST, 2015] C. P. S. P. W. G. NIST, "Draft framework for cyber-physical systems," National Institute of Standardards and Technology, Tech. Rep. Draft Release 0.8, September 2015. [cited at p. 13, 82]

[Odersky, 2006] M. Odersky, "A brief history of scala," http://www.artima.com/weblogs/viewpost.jsp?thread=163733, June 2006, (Visited on 09/22/2015). [cited at p. 25, 82]

[Odersky and al., 2004] M. Odersky and al., "An overview of the scala programming language," EPFL Lausanne, Switzerland, Tech. Rep. IC/2004/64, 2004. [cited at p. 82, 93]

[Osherove, 2009] R. Osherove, *The Art of Unit Testing: With Examples in .Net*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2009. [cited at p. 66, 82]

[Peierls et al., 2005] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2005. [cited at p. 25, 82]

[Prokopec, 2015] A. Prokopec, "Design of parallel collection's tasksupport - google groups," https://groups.google.com/forum/#!topic/scala-internals/lzxVlexkqEg, July 2015, (Visited on 12/10/2015). [cited at p. 82, 110]

[Prokopec and Miller, 2015] A. Prokopec and H. Miller, "Parallel collections - overview - scala documentation," http://docs.scala-lang.org/overviews/parallel-collections/overview.html, 2015, (Visited on 12/10/2015). [cited at p. 82, 111]

[Prokopec et al., 2011] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6853, pp. 136–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23397-5_14 [cited at p. 31, 83]

[Reiser, 1991] M. Reiser, *The Oberon System: User Guide and Programmer's Manual*. New York, NY, USA: ACM, 1991. [cited at p. 64, 83]

[Reisinger, 2014] D. Reisinger, "Keeping up with moore's law proves difficult for intel - cnet," http://www.cnet.com/news/keeping-up-with-moores-law-proves-difficult-for-intel/, July 2014, (Visited on 12/18/2015). [cited at p. 63, 83]

[Rochester, 1955] N. Rochester, "The computer and its peripheral equipment," in *Papers and Discussions Presented at the the November 7-9, 1955, Eastern Joint AIEE-IRE Computer Conference: Computers in Business and Industrial Systems*, ser. AIEE-IRE '55 (Eastern). New York, NY, USA: ACM, 1955, pp. 64–69. [Online]. Available: http://doi.acm.org/10.1145/1455319.1455330 [cited at p. 23, 83]

[SISO, 1999] S. I. S. O. SISO, "Siso-ref-002-1999: Fidelity implementation study group report," https://www.sisostds.org/ProductsPublications/ReferenceDocuments.aspx, 1999, (Visited on 12/18/2015). [cited at p. 13, 83]

[Sommers, 2008] F. Sommers, "Scala tendencies and concurrency," http://www.artima.com/lejava/articles/javaone_2008_martin_odersky.html, May 2008, (Visited on 10/23/2015). [cited at p. 73, 83]

[Sun et al., 2011] Y. Sun, S. Vogel, H. Steuer, A. Siemens, and E. Sector, "Combining advantages of specialized simulation tools and modelica models using functional mock-up interface (fmi)," in *Proceedings of the 8th MODELICA Conference*, 2011. [cited at p. 16, 17, 83]

[Typesafe Inc, 2015] Typesafe Inc, "Akka scala documentation," http://akka.io/docs/, Akka, September 2015, Release 2.4.0. [cited at p. 30, 83, 111]

[van Acker et al., 2015] B. van Acker, J. Denil, H. Vangheluwe, and P. D. Meulenaere, "Generation of an Optimised Master Algorithm for FMI Co-simulation," in *DEVS '15 Proceedings of the Symposium on Theory of Modeling & Simulation*, January 2015. [cited at p. 14, 83]

[Wirth, 1995] N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995. [cited at p. 64, 83]

Appendices

Appendix A

FMU State

This appendix contains a diagram and a table of the state machine used for Co-Simulation in the Functional Mock-up Interface (FMI) standard.

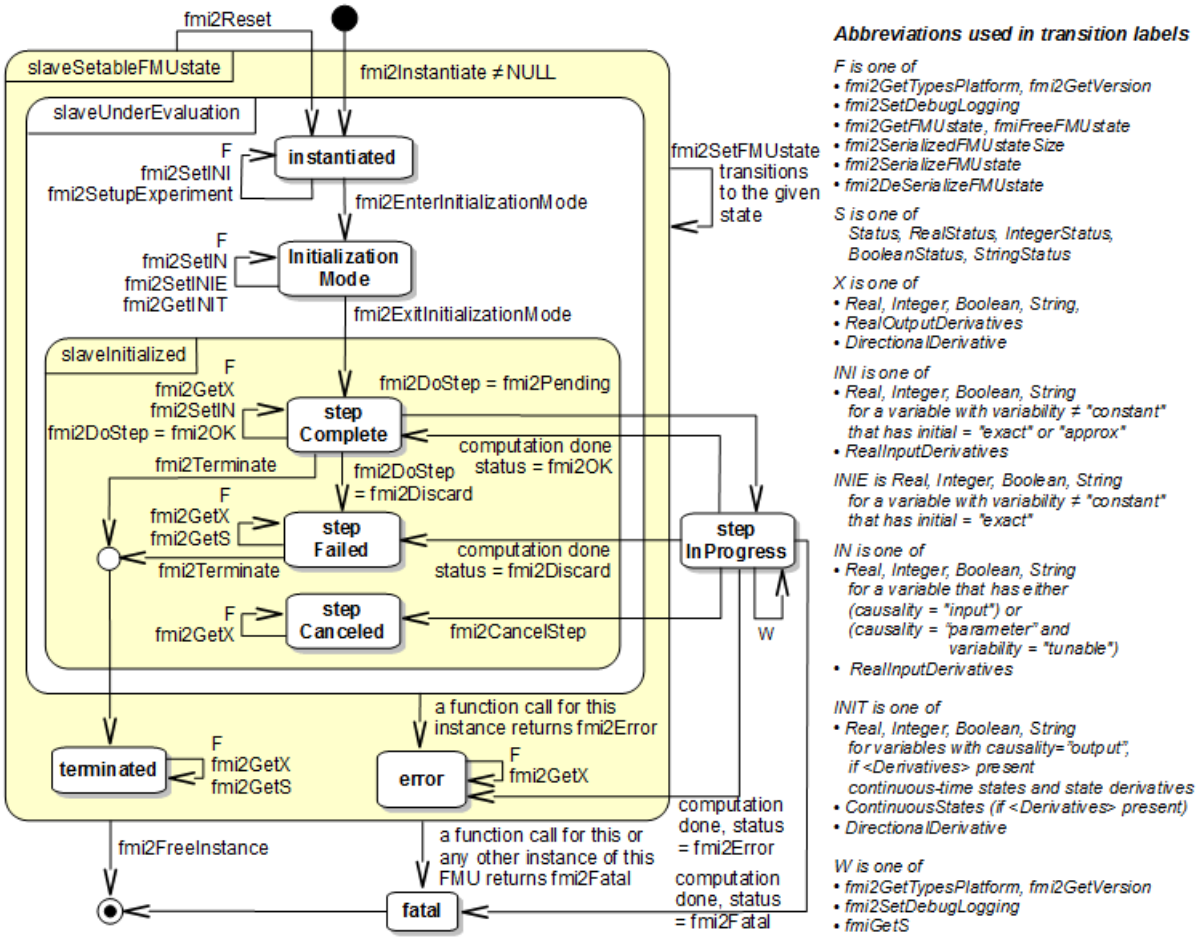


Figure A.1: Co-Simulation state machine for an Functional Mock-up Unit (FMU) [FMI development group, 2014a].

Function	FMI 2.0 forCo-Simulation									
	start, end	instantiated	Initialization Mode	stepComplete	stepInProgress	stepFailed	stepCanceled	terminated	error	fatal
fmi2GetTypesPlatform	x	x	x	x	x	x	x	x	x	
fmi2GetVersion	x	x	x	x	x	x	x	x	x	
fmi2SetDebugLogging		x	x	x	x	x	x	x	x	
fmi2Instantiate	x									
fmi2FreeInstance		x	x	x		x	x	x	x	
fmi2SetupExperiment		x								
fmi2EnterInitializationMode		x								
fmi2ExitInitializationMode			x							
fmi2Terminate				x		x				
fmi2Reset		x	x	x		x	x	x	x	
fmi2GetReal			2	x		8	7	x	7	
fmi2GetInteger			2	x		8	7	x	7	
fmi2GetBoolean			2	x		8	7	x	7	
fmi2GetString			2	x		8	7	x	7	
fmi2SetReal		1	3	6						
fmi2SetInteger		1	3	6						
fmi2SetBoolean		1	3	6						
fmi2SetString		1	3	6						
fmi2GetFMUstate		x	x	x		8	7	x	7	
fmi2SetFMUstate		x	x	x		x	x	x	x	
fmi2FreeFMUstate		x	x	x		x	x	x	x	
fmi2SerializedFMUstateSize		x	x	x		x	x	x	x	
fmi2SerializeFMUstate		x	x	x		x	x	x	x	
fmi2DeSerializeFMUstate		x	x	x		x	x	x	x	
fmi2GetDirectionalDerivative			x	x		8	7	x	7	
fmi2SetRealInputDerivatives		x	x	x						
fmi2GetRealOutputDerivatives				x		8	x	x	7	
fmi2DoStep				x						
fmi2CancelStep					x					
fmi2GetStatus				x	x	x		x		
fmi2GetRealStatus				x	x	x		x		
fmi2GetIntegerStatus				x	x	x		x		
fmi2GetBooleanStatus				x	x	x		x		
fmi2GetStringStatus				x	x	x		x		

Figure A.2: Co-Simulation state table for an FMU [FMI development group, 2014a].

Meaning of symbols and numbers in figure A.2.

x means: call is allowed in the corresponding state
number means: call is allowed if the indicated condition holds:
1 for a variable with variability $\neq$ ”constant“ that has initial = ”exact“ or ”approx“
2 for a variable with causality = ”output“ or continuous-time states or state derivatives (if element <Derivatives> is present)
3 for a variable with variability $\neq$ ”constant” that has initial=”exact“ or causality=”input“
6 for a variable with causality = ”input“ or (causality = ”parameter“ and variability = ”tunable“

7 always, but retrieved values are usable for debugging only
8 always, but if status is other than fmi2Terminated, retrieved values are useable for debugging only.

Appendix B

FMI formalization overview sheet

The formulas and descriptions below is a condensed version of the proposed formalization of the FMI [Broman et al., 2013].

$\mathbf{C}$ Set of all FMU instances in a model
: Set of all FMU instances coordinated by the same MA

$\mathbf{c} \in \mathbf{C}$ FMU Instance modifier
: One FMU instance, c, is an element in C

$\mathbf{S_c}$ Set of state valuations for instance c
: Given an instance c, S_c denotes the set of all possible states that c may be in

$\mathbf{U_c}$ Set of input port variables for instance c

$\mathbf{Y_c}$ Set of output port variables for instance c

$\mathbb{V}$ Set of values that a variable may take on
: Assume single universe of values for all variables

$\mathbf{D_c} \subseteq \mathbf{U_c} \times \mathbf{Y_c}$ I/O dependency for instance c
: The XML file in an FMU can (optionally) express the dependencies between input and output variables of an FMU. The set of all such input/output dependencies of an FMU instance c are modeled as a binary relation. $(u, y) \in D_c$ means output y of c is directly dependent on input u of c

$\mathbf{U} = \cup_{\mathbf{c} \in \mathbf{C}} \mathbf{U_c}$ Set of all input variables in a model

$\mathbf{Y} = \cup_{\mathbf{c} \in \mathbf{C}} \mathbf{Y_c}$ Set of all output variables in a model

$\mathbf{D} = \cup_{\mathbf{c} \in \mathbf{C}} \mathbf{D_c}$ Set of all I/O dependencies in a model

$\mathbf{P : U \rightarrow Y}$ Port mapping
: Connections between FMU instances in the model. U and Y are the sets of all input and output variables of all instances, respectively. P is a total function, that maps every input to a unique output variable. It is assumed, that the model is closed. Multiple inputs can have the same output, yet one input can only lead to one output by the definition of function.

Functions

$\mathbf{init_c : \mathbb{R}_{\geq 0}}$ Corresponds to fmiInitializeSlave.
Initializes FMU instance c with given start time t

$\mathbf{set_c : S_c \times U_c \times \mathbb{V} \to S_c}$ Corresponds to fmiSetXXX.
Given FMU instance c, given current state $s \in S_c$, input variable $u \in U_c$, and value $v \in \mathbb{V}$ returns the new state of c obtained by setting u to v and keeping the rest unchanged.

$\mathbf{get_c : S_c \times Y_c \to \mathbb{V}}$ Corresponds to fmiGetXXX.
Returns the value of output variable y of FMU instance c at state s

$\mathbf{doStep_c : S_c \times \mathbb{R}_{\geq 0} \to S_c \times \mathbb{R}_{\geq 0}}$ Corresponds to doStep.
Takes an input the current state s of FMU instance c and a non-negative real value $h \in \mathbb{R}_{\geq 0}$, corresponding to the communicationStepSize.

FMU contract

A0: If $\mathbf{doStep_c(s, h) = (s', h')}$ then $\mathbf{0 \leq h' \leq h}$

A1: If $\mathbf{doStep_c(s, h) = (s', h')}$ then for any h" where $\mathbf{0 \leq h'' \leq h'}$, $\mathbf{doStep_c(s, h'') = (s'', h'')}$ for some s"
If an FMU accepts a certain time step h, or at least makes partial progress until $h' \leq h$ then it must accept any time step $h'' \leq h'$

$\mathbf{s' = s[u := v]}$ Given state $s \in S_c$ of some instance $c \in C$ and given input variable $u \in U_c$ and value $v \in \mathbb{V}$ we denote by $s' = s[u := v]$ the state that is identical til s' except that s' assigns value v to variable u

A2: Let $\mathbf{s' = set_c(s, u, v)}$. Then $\mathbf{s' = s[u := v]}$

A3: Let $\mathbf{v = get_c(s, y)}$ and $\mathbf{v' = get_c(s', y)}$. If $\mathbf{s' = s[u_1 := v_1, ..., u_k := v_k]}$ and output variable y does not directly depend on any input $\mathbf{u_1, ..., u_k}$, then $\mathbf{v' = v}$

Determinate execution

(1): $\mathbf{getMaxStepSize_c : S_c \to \mathbb{R}_{\geq 0} \cup \{\infty\}}$
Returns an upper bound on the step size that the FMU can accept.

A4: If $\mathbf{c \in C_P}$ and $\mathbf{s \in S_c}$ and $\mathbf{getMaxStepSize_c(s) = h}$ then for all $\mathbf{h'}$ where $\mathbf{0 \leq h' \leq h}$, $\mathbf{doStep_c(s,' h) = (s', h')}$ for some s'.
$\mathbf{C_P}$ Is the set of FMU instances that implement (1).
An instance in C_P will accept any time step smaller than or equal to the time step returned by getMaxStepSize.

A5: (a): $\mathbf{|C_L| \leq 1}$

(b): $\mathbf{|C_L| \cup C_R \cup C_P = C}$
$\mathbf{C_R}$ set of FMU isntances with rollback capability, i.e. supports setting and getting states
$\mathbf{C_L}$ set of FMU instances that are not in $\mathbf{C_R}$ and not in $\mathbf{C_P}$

(c): $\mathbf{C_L \cap C_R = \emptyset}$ and $\mathbf{C_R \cap C_P = \emptyset}$ and $\mathbf{C_P \cap C_L = \emptyset}$

Appendix C

Scala class hierarchy

This appendix contains the Scala class hierarchy shown in figure C.1.

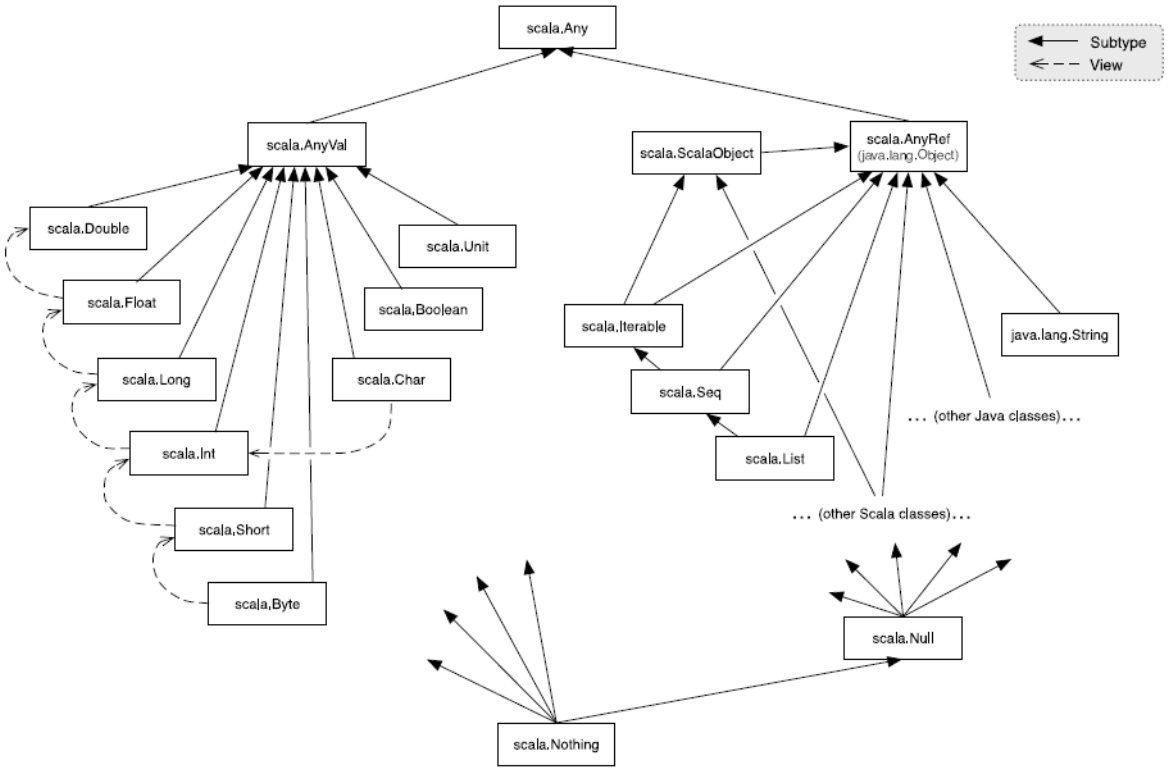


Figure C.1: Scala class hierarchy [Odersky and al., 2004].

Appendix D

Actor Trait

The trait used in the implementation using actors described in section 3.3.4 is shown in listing D.1.

Scala

```
1 trait FmuActor {
2   def doSimulationStep(mi: ModelInstance, si:
      ↪ FmiSimulationInstance2, currentCommunicationPoint:
      ↪ Double, communicationStepSize:
3   Double, inputState: Option[InputState], rb: Option[
      ↪ IFmiComponentState]):
4   Future[Either[(ModelInstance, (InstanceState, Option[
      ↪ IFmiComponentState])), Either[(ModelInstance, (
      ↪ Double, Option[IFmiComponentState])), (
      ↪ ModelInstance,
5   Fmi2Status)] with Product with Serializable] with Product
      ↪  with Serializable]
6 }
```

Listing D.1: `FmuActor` trait

Appendix E

Test Framework

This appendix contains the code of the test framework. The test framework was used to evaluate performance and correctness of Co-Simulation Orchestration Engines (COEs) implementations. It was described in chapter 4. The code for the class `TestRunner` is shown in listing E.1, the code for the class `CoeLauncher` is shown in listing E.2, the code for the class `SimulationResult` is shown in listing E.3, the code for the class ServerRequester is shown in listing E.4, and the code for the class `IOThreadHandler` is shown in listing E.5. An example of the test HVAC Test1 described in section 4.4.1 that uses the test framework is shown in listing E.6

```
1  public class TestRunner {
2    public static SimulationResults RunTestsForJar(String
        ↪ overallTestName, String jarPath, String configPath,
        ↪ double endTime, List<String> arguments, String baseline)
        ↪  throws Exception {
3      Map<String, String> results = new HashMap<>();
4      Map<String,Long> executionTimes = new HashMap<>();
5      Map<String,Boolean> baselineNonMatches = new HashMap<>();
6      if(arguments != null &&  arguments.size() > 0) {
7        for (String argument : arguments) {
8          // do some work here on intValue
9          RunSimulation(overallTestName + " - " + argument,
              ↪ jarPath, configPath, endTime, argument.split(" "),
              ↪  baseline, results, executionTimes,
              ↪ baselineNonMatches);  }
10     } else {
11       RunSimulation(overallTestName, jarPath, configPath,
            ↪ endTime, null, baseline, results, executionTimes,
            ↪ baselineNonMatches);
12     }
13     return new SimulationResults(results, executionTimes,
          ↪ baselineNonMatches);
14   }
15   private static void RunSimulation(String overallTestName,
        ↪ String jarPath, String configPath, double endTime,
        ↪ String[] argument,
```

```
16    String baseline, Map<String,
17    String> results, Map<String, Long> executionTimes, Map<String,
          ↪  Boolean> doesNotMatchBaseline) throws Exception {
18      SimulationResult result = RunSimulation(endTime,jarPath,
            ↪ configPath,argument);
19      String name = overallTestName;
20      if(baseline != null && result.result.equals(baseline) ==
            ↪ false){
21        doesNotMatchBaseline.put(name, true);}
22      } else{
23        doesNotMatchBaseline.put(name, false);
24      }
25      results.put(name,result.result);
26      executionTimes.put(name,result.executionTime);
27    }
28
29    public static SimulationResult RunSimulation(double endTime,
          ↪ String jarPath, String configPath, String[] arguments)
          ↪ throws Exception {
30      Process proc = CoeLauncher.launchCoe(jarPath, arguments);
31      //Keep printing inputs from the java process to the console
32      ThreadHelpers.IOThreadHandler outputHandler = new
            ↪ ThreadHelpers.IOThreadHandler(
33      proc.getInputStream());
34      outputHandler.start();
35      //Initialize the CoE
36      JsonNode initializationDataJson = ServerRequester.
            ↪ InvokeInitialize(configPath);
37      int sessionId = initializationDataJson.path("sessionId").
            ↪ asInt();
38      //Simulate
39      JsonNode simulationDataJson = ServerRequester.InvokeSimulate
            ↪ (endTime, sessionId);
40      String status = simulationDataJson.path("status").asText();
41      Long executionTime = simulationDataJson.path("lastExecTime")
            ↪ .asLong();
42      System.out.println("Status: " + status);
43      System.out.println("Execution time: " + executionTime);
44      //Get results
45      String resultData = ServerRequester.invokeResult(sessionId);
46      //Terminate the CoE
47      proc.destroy();
48      //Return the results
49      return new SimulationResult(resultData, executionTime);
50    }
51  }
```

Listing E.1: Java. The code for the class `TestRunner`

```java
public class CoeLauncher {
  //Returns once the CoE has started.
  public static Process launchCoe(String fullPath, String[]
     ↪ arguments) throws IOException {
    String argument = "";
    if(arguments != null && arguments.length > 0){
    argument = " " + String.join(" ", arguments);
    }
    Process proc = Runtime.getRuntime().exec("java -jar " +
       ↪ fullPath + argument);
    BufferedReader br = new BufferedReader(new InputStreamReader
       ↪ ((proc.getInputStream())));
    String output;
    while ((output = br.readLine()) != null) {
    System.out.println(output);
    if (output.equalsIgnoreCase("Hit Enter To Stop."))
    break;
    }

    return proc;
  }
  public static Process launchCoe(String fullPath) throws
     ↪ IOException {
    return launchCoe(fullPath, null);
  }
}
```

Listing E.2: Java. The code for the class `CoeLauncher`

```java
public class SimulationResults {
  public Map<String, Boolean> baselineNonMatches;
  public Map<String, String> results;
  public Map<String,Long> executionTimes;
  public SimulationResults(Map<String, String> results, Map<
     ↪ String,Long> executionTimes){
    this.results = results;
    this.executionTimes = executionTimes;
  }

  public SimulationResults(Map<String, String> results, Map<
     ↪ String, Long> executionTimes, Map<String, Boolean>
     ↪ baselineMatches) {
    this.results = results;
    this.executionTimes = executionTimes;
    this.baselineNonMatches = baselineMatches;
  }
}
```

Listing E.3: Java. The code for the class `SimulationResults`

```
1  public class ServerRequester {
2    final static ObjectMapper mapper = new ObjectMapper();
3
4    public static JsonNode PerformPostRequest(byte[] data, URL url
       ↪ ) throws Exception {
5      HttpURLConnection urlConn;
6      urlConn = (HttpURLConnection) url.openConnection();
7      urlConn.setDoOutput(true); //Makes it a POST request
8      urlConn.addRequestProperty("Content-Type", "application/json
         ↪ ");
9      urlConn.setRequestProperty("Content-Length", Integer.
         ↪ toString(data.length));
10     urlConn.getOutputStream().write(data);
11
12     BufferedReader br = new BufferedReader(new InputStreamReader
         ↪ ((urlConn.getInputStream())));
13     StringBuilder sb = new StringBuilder();
14     String output;
15     while ((output = br.readLine()) != null) {
16     sb.append(output);}
17
18     urlConn.disconnect();
19
20     //Removing [] around json data
21     String response = sb.substring(1, sb.length()-1);
22     return mapper.readTree(response);
23   }
24
25   public static JsonNode InvokeInitialize(String configFullPath)
       ↪  throws Exception {
26     byte[] data = Files.readAllBytes(Paths.get(configFullPath));
27     URL mUrl = new URL("http://localhost:8082/initialize");
28
29     return PerformPostRequest(data, mUrl);
30   }
31
32   public static JsonNode InvokeSimulate(double endTime, int
       ↪ sessionId) throws Exception {
33     Map<String,Double> startEndTime = new HashMap<>();
34     startEndTime.put("startTime",0.0);
35     startEndTime.put("endTime",endTime);
36     String startEndTimeJson = new ObjectMapper().
         ↪ writeValueAsString(startEndTime);
37     byte[] data = startEndTimeJson.getBytes();
38
39     URL url = new URL("http://localhost:8082/simulate/"+
         ↪ sessionId);
40
```

```
41      return PerformPostRequest(data, url);
42    }
43
44    public static String invokeResult(int sessionId) throws
          ↪ Exception {
45      HttpURLConnection urlConn;
46      URL mUrl = new URL("http://localhost:8082/result/"+sessionId
            ↪ );
47      urlConn = (HttpURLConnection) mUrl.openConnection();
48      urlConn.addRequestProperty("Content-Type", "text/plain");
49
50      BufferedReader br = new BufferedReader(new InputStreamReader
            ↪ ((urlConn.getInputStream())));
51      ArrayList<String> outputArray = new ArrayList<>();
52      String output;
53      while ((output = br.readLine()) != null) {
54        outputArray.add(output);
55      }
56
57      urlConn.disconnect();
58
59      //Csv format
60      return String.join("\n", outputArray);
61    }
62  }
```

Listing E.4: Java. The code for the class `ServerRequester`

```
1   public class ThreadHelpers {
2     public static class IOThreadHandler extends Thread {
3       private InputStream inputStream;
4       private StringBuilder output = new StringBuilder();
5
6       IOThreadHandler(InputStream inputStream) {
7         this.inputStream = inputStream;
8       }
9
10      public void run() {
11        Scanner br = null;
12        try {
13        br = new Scanner(new InputStreamReader(inputStream));
14        String line = null;
15        while (br.hasNextLine()) {
16          line = br.nextLine();
17          System.out.println(line);
18          output.append(line
19          + System.getProperty("line.separator"));
20        }
21        } finally {
```

```
22          br.close();
23        }
24      }
25
26      public StringBuilder getOutput() {
27        return output;
28      }
29    }
30  }
```

Listing E.5: Java. The code for the class `IOThreadHandler`

```
1  public void HVAC1Controller4VCU01Step1000EndTime() throws
       ↪ Exception {
2    String configPath = "C:\\thesis\\source\\testing\\
       ↪ hvac4fcu01step.json";
3    double endTime = 1000.0;
4    int nrOfSimulationsPerWaitTime = 5;
5    SimulationResult baselineResult = null;
6
7    ArrayList<Tuple<String, String>> jars = new ArrayList<>();
8    jars.add(new Tuple<>("Baseline", "C:\\thesis\\source\\testing
       ↪ \\jars\\baseline.jar"));
9    jars.add(new Tuple<>("Future", "C:\\thesis\\source\\testing\\
       ↪ jars\\fullFuture.jar"));
10   jars.add(new Tuple<>("Par", "C:\\thesis\\source\\testing\\jars
       ↪ \\fullPar.jar"));
11   jars.add(new Tuple<>("Actor", "C:\\thesis\\source\\testing\\
       ↪ jars\\fullActor.jar"));
12   //Create directory to contain test result
13   File directory = new File(TestUtils.getCurrentTimeStamp());
14   directory.mkdir();
15
16   //Create and open file to contain test results
17   File executionResults = new File(directory, "executionTimes.
       ↪ csv");
18   FileWriter fStream = new FileWriter(executionResults, true);
19   BufferedWriter bufferedWriterExecResults = new BufferedWriter(
       ↪ fStream);
20   PrintWriter executionResultWriter = new PrintWriter(
       ↪ bufferedWriterExecResults);
21   executionResultWriter.println("baseline,Future,Par,Actor");
22   ArrayList<String> csvResults = new ArrayList<>();
23
24   for (Tuple<String, String> tuple : jars) {
25   ArrayList<Long> execTimes = new ArrayList<>();
26   for (int j = 1; j <= nrOfSimulationsPerWaitTime; j++) {
27   SimulationResult result = TestRunner.RunSimulation(endTime,
       ↪ tuple.y, configPath, null);
```

```
28    if(tuple.x == "Baseline" && baselineResult == null)
29      baselineResult = result;
30    else
31      Assert.assertTrue("Failed with " + tuple.x, result.result.
          ↪ equals(baselineResult.result));
32    execTimes.add(result.executionTime);
33    }
34    if(nrOfSimulationsPerWaitTime >= 3)
35    {execTimes.remove(execTimes.indexOf(Collections.min(execTimes)
          ↪ ));
36    execTimes.remove(execTimes.indexOf(Collections.max(execTimes))
          ↪ );
37    }
38    csvResults.add(String.valueOf(TestUtils.sum(execTimes) / (
          ↪ nrOfSimulationsPerWaitTime-2)));
39    }
40
41    executionResultWriter.println(String.join(",", csvResults));
42
43    executionResultWriter.close();
44    System.out.println("Stored data in " + executionResults.
          ↪ getAbsolutePath());
45  }
```

Listing E.6: Java. Example of the test HVAC Test1, that is described in section 4.4.1

Appendix F

Copy-script

The script in listing F.1 copies an Functional Mock-up Unit (FMU), modifies the FMU configuration file by changing the model name and guid along with modifying the COE configuration file to make use of the new FMUs. To better understand the operations in the script, section 2.4.1 describes an FMU package. The script was mentioned in section 4.3.

```
1  #!/usr/bin/env bash
2  #Name of output without guid
3  JsonOutputProperty=output
4  #Name of input without guid
5  JsonInputProperty=input
6  #Path to the FMU to chain
7  FmuPath=$1
8  #"fmus/performance/integrate"
9  #cat ${FmuPath}/modelDescription.xml
10 echo "${FmuPath}"
11 #FMU base name taken from directory
12 FmuBaseName=$(basename $FmuPath)
13 #Destination folder for the new FMUs
14 FmuDestFolder=$2
15 #"fmus/performance/test"
16 #Basepath to insert in FMUs without FMU name
17 ConfigFmuBasePath=$3
18 #../fmus/performance/
19 #Path to initial config file
20 ConfigJsonPath=$4
21 #Types of libraries
22 LibraryTypes=("dll" "dylib" "so")
23 #Guid of the FMU
24 #FmuGuid=$(sed -n 's/.*guid="{\([^"]\+\)}.*/\1/p' $FmuPath/
   ↪ modelDescription.xml)
25 FmuGuid=`grep -oh 'guid="{\([^"]\+\)}"' "${FmuPath}/
   ↪ modelDescription.xml" | sed 's|guid=||g' | tr -d '"' | tr
   ↪ -d '{' | tr -d '}'`
26 #FmuGuid=`echo '${FmuPath}/modelDescription.xml'`
```

```
27 echo $FmuGuid
28
29 echo Create output config
30 cp $ConfigJsonPath performance.json
31 ConfigJsonPath=performance.json
32 mkdir -p $FmuDestFolder
33
34 END=$5
35 for i in $(seq "$END")
36 do
37
38 #Copy the existing FMU to a new directory
39 NewFmuPath="${FmuDestFolder}/$(basename $FmuPath)$i"
40 echo "BLAA $NewFmuPath"
41 rm -rf $NewFmuPath
42 cp -r $FmuPath $NewFmuPath
43 echo "Copied FMU from $FmuPath to $NewFmuPath"
44
45 #Change the library names so e.g. integrate.dll becomes
      ↪ integrate1.dll
46 for type in ${LibraryTypes[@]}
47 do
48 for lib in $(find $NewFmuPath -iname '*.'"$type")
49 do
50 NewLibFileName=$(dirname $lib)/${FmuBaseName}${i}.${lib##*.}
51 echo Renaming ${lib} to ${NewLibFileName}
52 mv  $lib $NewLibFileName
53 done
54 done
55
56 #Replace the old GUID with the new guid in the modelDescription.
      ↪ xml file
57 newGuid="$FmuGuid$i"
58 sed -i.kk "s/$FmuGuid/$newGuid/g" "$NewFmuPath/modelDescription.
      ↪ xml"
59 echo "Replaced guid in modelDescription.xml with $newGuid"
60
61 #New FMU in config.json
62 FmuConfigAppend=".fmus |= (.+ [\"
      ↪ $ConfigFmuBasePath$FmuBaseName$i\"] | unique)"
63
64
65 #New connection in config.json
66 if [ $i != 1 ]; then
67 CONGUID="$FmuGuid"$(($i-1))
68 else
69 CONGUID=$FmuGuid
70 fi
71
```

```
72 #Create the new config.json file
73 FmuConnectionAppend=".connections.\"{$CONGUID}.int1.
      ↪ $JsonOutputProperty\" = [\"{$newGuid}.int1.
      ↪ $JsonInputProperty\"]"
74 "jq" "$FmuConnectionAppend | $FmuConfigAppend" "$ConfigJsonPath"
      ↪  > "$newGuid.json"
75 mv -f "$newGuid.json" "$ConfigJsonPath"
76 echo "Appended the new FMU to fmus and connections in config.
      ↪ json"
77 done
78 exec $SHELL
```

Listing F.1: Bash. The code for the shell script that copies an FMU.

Appendix G

Evaluation of Quality Attributes

This appendix describes the evaluation of the quality attributes composability, simplicity, configurability, scalability, documentation, and performance described in section 4.6. The rating scheme is described in table G.1

Property	**Rating**		
	-	**+**	**++**
Composability	No composability	Limited composability	Easy composability
Simplicity	Extensive initialization and complicated usage	Some initialization and moderate usage	Basic initialization and easy usage
Configurability	Not configurable	Moderately configurable	Very configurable
Scalability	No functionality for scaling	Some scaling functionality	Extensive scaling functionality
Documentation	Limited documentation	Basic documentation	Well documented and many resources
Performance	Slowest	In the middle	Fastest

Table G.1: Rating of quality attributes. Performance is relative to the three implementations.

Composability

Parallel Collections: Parallel collections invokes functions concurrently on a range of collections as shown in section 2.8.3 and gathers the results in a blocking fashion. The composability of parallel collections is therefore given the rating **-**.

Futures: Futures offers various possibilities for composability, some of them are shown in section 2.8.1: for-comprehensions, sequence, callbacks, combinators and so forth. With parallel collections it is necessary to wait for the entire computation to finish, whereas the result of each future can be accessed as soon as it is completed. The composability of futures is therefore given the rating **+**.

Actors: Actors can be made to return futures as shown in section 2.8.2, and therefore offers the composability features of futures. Actors also provide additional composable features such as hierarchical structures, they can send messages to other actors etc. The composability of actors is therefore given the rating **++**.

Simplicity

Parallel Collections: Parallel collections can be taken advantage of by invoking the function `par` on a range of collections, and can be converted back to a regular collection by invoking the function `toSeq`. It is simple to use and requires no initialization. The simplicity of parallel collections is therefore given the rating **++**.

Futures: Futures can be taken advantage of by wrapping the functions that are to run concurrently in future objects. They require an `ExecutionContext`, which is passed as an *implicit parameter*. This means, that an `ExecutionContext` must be passed explicitly to the function or it is provided automatically [EPFL, 2015]. The passing of an `ExecutionContext` and the wrapping of functions in future objects are considered to be conceptually simple. The simplicity of futures is therefore given a rating of **++**.

Actors: Actors require the external library Akka and a central initialization of the actor system. The actor system is used for constructing actors, and this is also considered initialization. As the actors are objects, they have to be passed around as parameters for functions to use them. The simplicity of actors therefore get the rating **+**.

Configurability

Parallel Collections: Parallel collections uses task support, which is backed by an `ExecutionContext`. However, configuring task support makes *chaining* collection functions impossible as shown in listing G.1. It shows, that in order to perform a map function on the parallel collection with a different task support than the default, it is necessary to introduce mutable state and split the chaining of functions. This example: `Array(1,2,3).par.map(x => x)` uses the default task support, but supports chaining of functions [1]. So parallel collections are configurable to an extent, but it comes with trade offs. The configurability of parallel collections is therefore given the rating **+**.

Scala

```scala
1       val array = Array(1,2,3).Par
2       array.taskSupport = new ForkJoinTaskSupport(new scala
          ↪ .concurrent.forkjoin.ForkJoinPool(2))
3       array.map(x => x)
```

Listing G.1: Configuration of parallel collections

Futures: Futures use *implicit parameters* as mentioned previously. Thereby an `ExecutionContext` can be used just by importing it without having to change the implementation using futures.

[1] Discussions regarding this issue has been ongoing, and it might be changed in a future version of Scala [Prokopec, 2015]

Futures are configurable because of their usage of an `ExecutionContext`. The configurability of futures is therefore given the rating **+**.

Actors: Actors can be configured with dispatchers, which contains execution contexts as used by parallel collections and futures. However the dispatchers contain additional functionality, e.g. throughput, which is a measure of how many messages an actor should process before the thread jumps to the next actor. Besides the additional functionality in dispatchers, it is also possible to configure logging, routing, remoting, etc. The configurability of actors is therefore given the rating **++**.

Scalability

Parallel Collections: Parallel collections can take advantage of concurrency, but otherwise it does not offer anything in terms of scalability, as it performs blocking operations. The scalability of parallel collections is therefore given the rating **-**.

Futures: Because futures provide the opportunity of performing non-blocking operations it can increase the scalability of the COE. As the COE runs as a web server, it is undesirable to block the thread responsible of handling requests, and therefore futures can be used to perform simulations without leaving the server unresponsive until the simulation is finished. Similar to parallel collections it can also take advantage of concurrency. The scalability of futures is therefore given the rating **+**.

Actors: Actors offers the features of futures and additional features. It provides remoting capabilities, making it able to split a simulation workload not only across cores but across distributed hardware as well. It also offers session management capabilities, fault handling e.g. with supervisor actors as described in section 2.8.2 and so forth. The possibility of sending untyped messages makes it possible to provide a single api, that can redirect messages without having to know the content of them e.g. in case of load-balancing. The scalability of actors is therefore given the rating **++**.

Documentation

Parallel Collections: Official documentation exists on parallel collections [Prokopec and Miller, 2015]; however, there are not many resources available on configuring parallel collections using execution contexts. Documentation on parallel collections is therefore given the rating **+**.

Futures: Official documentation exists on futures [Haller et al.], and there are several resources available on using and configuring them. Documentation on futures is therefore given the rating **++**.

Actors: Official documentation exists on actors [Typesafe Inc, 2015], and there are several resources available on using and configuring them. These resources also include documentation on advanced scenarios such as session management, load balancing, fault tolerance etc. Documentation on actors is therefore given the rating **++**.

Casper Thule, Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS, 2016

Department of Engineering
Aarhus University
Inge Lehmanns Gade 10
8000 Aarhus
Denmark

Tel.: +45 8715 0000