FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Analysis of Coupling Strategies and Protocols for Co-Simulation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering/Internet Computing**

eingereicht von

**Clemens Pühringer, Bsc.**
Matrikelnummer 1026571

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Bernhard Heinzl

Wien, 6. Dezember 2017

| Clemens Pühringer | Wolfgang Kastner |
|---|---|

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Analysis of Coupling Strategies and Protocols for Co-Simulation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering/Internet Computing**

by

**Clemens Pühringer, Bsc.**
Registration Number 1026571

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Bernhard Heinzl

Vienna, 6th December, 2017

| Clemens Pühringer | Wolfgang Kastner |
|---|---|

# Erklärung zur Verfassung der Arbeit

Clemens Pühringer, Bsc.
Rothenbergstrasse 19/1, 4942 Gurten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Dezember 2017

Clemens Pühringer

# Kurzfassung

Computersimulationen werden immer komplexer. Kontinuierliche und diskrete Modelle benötigen zur Simulation von Grund auf verschiedene Ansätze. Derzeit existierende Simulationsanwendungen unterstützen meist nur eine der beiden Methoden und bieten keine oder nur sehr eingeschränkte Unterstützung für die andere. Mithilfe der Co-Simulation kann ein Modell in mehrere Teile aufgeteilt werden. Diese Teile können anschließend mit unterschiedlichen Methoden und Anwendungen simuliert werden, welche zur Laufzeit Daten austauschen. Den derzeit verfügbaren Herangehensweisen fehlt es an Flexibilität. Viele der existierenden Frameworks sind schwierig zu verwenden oder wurden nur für eine spezielle Anwendung erstellt. Um die Implementierung von Co-Simulationen einfacher zu gestalten, benötigt es leichter zu handhabende und flexiblere Werkzeuge.

Diese Arbeit präsentiert einen Vergleich verschiedener Kopplungsstrategien und Kommunikationsprotokolle für Co-Simulation. Zu diesem Zweck wurde ein Co-Simulations-Framework implementiert, das Matlab/Simulink und OpenModelica unterstützt. Daten zwischen den Simulationen werden mit dem Simple Object Access Protocol (SOAP) und dem OPC Unified Architecture (OPC UA) binary Protokoll ausgetauscht. Diese Protokolle wurden aufgrund ihrer hohen Flexibilität und Erweiterbarkeit gewählt. Mit beiden Ansätzen ist es möglich, strukturierte Daten einfach zu transportieren. Das Framework unterstützt desweiteren weak und dynamic Coupling-Strategien. Obwohl weak coupling in Co-Simulationen ungenauer ist, wird es derzeit am häufigsten verwendet. Dem liegt zugrunde, dass andere Kopplungsstrategien oft wesentlich schwieriger zu implementieren und zu verweden sind. Im Zuge dieser Arbeit werden die Genauigkeit und die Geschwindigkeit verschiedener Kopplungsstrategien verglichen.

Als Proof-of-Concept wurde ein bestehendes Modell zur Simulation industrieller Energieeffizienz verwendet. Das Modell wurde in einen thermischen Teil und einen maschinellen Teil aufgeteilt. Der thermische Teil wird in Matlab/Simulink simuliert und beinhaltet Energieversorgung und Wärmeausbreitung innerhalb von vier thermischen Zonen eines Gebäudes. Der maschinelle Teil wird in OpenModelica simuliert und beinhaltet Maschinen, die elektrische Energie in thermische Energie umwandeln und abgeben. Im Zuge der Co-Simulation tauschen diese beiden Simulationen elektrische Energie und Wärmeenergie miteinander aus. Dies zeigt den Effekt verschiedener Kopplungsstrategien anhand der Verzögerung des Energieaustausches.

Die Ergebnisse der Co-Simulation werden mit Ergebnissen eines Referenzmodells verglichen und zeigen eine zufriedenstellende Übereinstimmung. Die Übereinstimmung ist nicht exakt, zeigt aber, dass sich die Ergebnisse durch die Reduzierung der Makro-Schrittweite an die originalen Ergebnisse annähern. Desweiteren wird demonstriert, dass dynamic coupling bei gleicher Makro-Schrittweite wesentlich genauere Ergebisse liefern kann als weak coupling. Dies zeigt, dass dynamic coupling durchaus das Potential hat schneller als weak coupling zu agieren.

# Abstract

Computer simulations are becoming increasingly complex. Different simulation techniques are needed to correctly simulate systems that consist of both continuous and discrete components. Current simulation tools offer support for either one or the other, but only limited or no support for both at the same time. By using co-simulation, a model can be split into multiple parts, which can then be simulated by different tools and methods and exchange data at runtime. Current co-simulation approaches are either hard to use in a custom co-simulation setup or are only designed for a very specific use-case. More flexible tools are needed to simplify the separation of a model into multiple parts.

This thesis presents a comparison of different coupling methods and protocols for data exchange in a co-simulation setup. For this purpose, a new co-simulation framework was developed which supports Matlab/Simulink and OpenModelica simulations and exchanges data via the Simple Object Access Protocol (SOAP) and OPC Unified Architecture (OPC UA) binary protocol. These high-level protocols were chosen due to their flexibility. Both allow to transport structured information and can be extended easily. Also, the framework is designed to handle weak and dynamic coupling methods. In current co-simulation setups, weak coupling is usually preferred even though it is less accurate. It is mainly used because dynamic coupling takes longer to implement and are harder to use. The different coupling methods are compared to each other in terms of speed and accuracy.

As a proof-of-concept, an existing model for simulating industrial energy efficiency was split into two parts and simulated with the framework. The first part of the model consists of a building with thermal zones and an energy supply system. It is simulated in Matlab/Simulink and manages energy and heat distribution. The second part of the model manages machines that convert electrical energy into heat, it is simulated in OpenModelica. In the co-simulation, both model parts exchange energy and other information with each other and thereby demonstrate the effects of different coupling methods due to communication delays.

The results of the co-simulation are validated with existing results of a reference implementation and show a satisfactory outcome. While not exact, they demonstrate that the results of the co-simulation converge towards the original results when reducing the macro-step size. They also show that dynamic coupling methods are far more accurate than the widely used weak coupling methods and may even provide better results in less time.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

Simulation is an important part for many scientific and industrial fields. It allows to analyze and predict many properties of the physical world. Simulations are typically used in cases where real-world trials are expensive (e.g. aeronautics, fluid dynamics), or simply impossible (e.g. cosmology). The simulation used in this thesis is based upon a case study analyzing energy efficiency in industrial environments. The study investigates different scenarios and the resulting energy usage and distribution. With the help of these simulations, weak points can be identified and the energy efficiency of factories can be improved.

As models get bigger and more complex, co-simulation, where multiple simulation models and/or tools are coupled at runtime to exchange data, plays a more important role. With the help of co-simulation, models can be split into smaller parts, which enables multiple people to work on a model simultaneously. This allows for experts in their respective fields to create the different parts of the model, which can then be combined into the co-simulation. Furthermore, different solvers (algorithms) can be used to simulate each part of the co-simulation. This can be beneficial when different solvers are better suited or produce more accurate results for different parts of the model. Most importantly, splitting the model allows researchers to combine multiple modelling approaches within one simulation. This can be very beneficial in hybrid models, where one part is more suited for discrete simulation and another part is better suited for continuous simulation.

A drawback of co-simulation is that there is currently little standardization of interfaces between the model parts. Current co-simulation solutions are often tailor-made for specific setups and tools, and are not reusable for different models. Standardized interfaces would make it possible to use the different elements of the model in other co-simulations. An example of such a dependency is PowerNet [LA11] which uses co-simulation for Smart

Grid simulation. It uses ns-2 as a discrete simulator for the data network and Modelica as continuous simulator for the electrical network.

Other technologies exist which can be used by a larger variety of simulators, but are still very limited with respect to extendability, scalability and fault tolerance. One such technology is the *Building Controls Virtual Test Bed* (BCVTB) [Nou]. It can be used in many simulation tools, but offers only low-level socket connections between them. The semantics of the data which is transferred over such a connection has to be defined on a per-application basis. Another drawback of BCVTB is that it only allows for loose coupling and a fixed step size for the whole co-simulation [HHR13].

The dependency on tools and setup also means that the parts of the model can not easily be exchanged with other teams and researchers. Dividing the model into smaller pieces and combining them also leads to more overhead, as the exchange of results between simulations (and/or even different systems) takes time. Another important factor is that different coupling strategies vary in execution time and accuracy. All strategies can lead to numerical errors, but the most used coupling strategy today (weak coupling), where data is exchanged at certain intervals, is also the least accurate. This strategy is primarily used because it is fast and easy to implement, and oftentimes it is the only choice available.

## 1.2 Problem Statement

The first challenge of the thesis is to describe what kind of information needs to be exchanged between between the participants in a co-simulation. Finding a suitable data model is an important first step because the following steps all build upon it.

The next challenge is to find appropriate technologies for actually exchanging the information within the co-simulation setup. These technologies should be extendable, scalable, and also have good usability. The overall aim in using such technologies is to create a new co-simulation tool that is easy to use, easy to integrate into existing projects, and powerful/fast enough to handle real-world applications.

Different technologies offer different trade-offs for these characteristics. Technologies like SOAP will have a lot more overhead than BCVTB, but offer much more flexibility in terms of metadata and can thus be extended much more easily. OPC UA binary takes the middle ground, it has less overhead than SOAP but also offers less usability and extendability. Other technologies like JSON could also be a valid option in this endeavour.

In addition to improvements in usability, we want to investigate different co-simulation strategies. The two basic co-simulation strategies are weak and dynamic coupling. The most popular today is weak coupling, where the simulations exchange data at specified timesteps. It is a very straight-forward approach and is feasible for values which do not vary a lot over time. The second strategy is dynamic coupling, this approach aims to

minimize the error by simulating a timestep multiple times with updated values. This approach will usually take significantly longer than weak coupling due to the fact that a time step is simulated more than once.

The questions that this thesis aims to answer are which technologies are best suited for co-simulation based on their functional and non-functional characteristics. Specifically, are SOAP and OPC UA suitable technologies to be used in co-simulation and the different coupling strategies? And if so, how do they compare to existing technologies with regard to usability, extendability and scalability?

## 1.3 Goal

The goal of this thesis is to research and test different technologies for their use in co-simulation. This is done in two main parts.

The first part is the proof-of-concept implementation of a co-simulation software which will be able to execute co-simulations between Simulink and OpenModelica simulations. Simulink and OpenModelica are chosen based on the case study which is used as a basis for the models for the co-simulation. The case study is about interdisciplinary investigations of energy efficiency in production facilities, and can easily be separated into multiple models for a co-simulation. This makes it an ideal candidate for this thesis. Existing models for the case study are implemented in Matlab and OpenModelica, therefore it is reasonable to also use these tools in the new co-simulation software. Two technologies, SOAP and OPC UA, will be used in the co-simulation software, where SOAP will be used in conjunction with Simulink simulations, and OPC UA will be used for OpenModelica simulations. The software will also support weak and dynamic coupling strategies. As dynamic coupling is not used very often in current co-simulation setups, it presents an additional opportunity to analyze the impact in simulation time and the accurracy of the results based on varying parameters.

The second part is an analysis of the results produced by the proof-of-concept implementation. The goal of this analysis is an overview of properties (usability, scalability, extendability) that indicate which technologies are the most promising in future applications. To validate the correctness of the results, the existing results of the case study will be used to compare against the results of the thesis.

## 1.4 Method

In a first step, a literature study will be conducted to gain insight into how co-simulation is generally executed and how different coupling strategies work. Additionally, the impact of the different protocols and associated data formats on the performance of the system will be evaluated.

In the next step, the requirements for the co-simulation system will be defined based on the findings of step one. The requirements will include non-functional requirements like usability, scalability and extendability as well as functional requirements for the system. The functional requirements will define which features will be supported by the finished product, like the ability to perform weakly and dynamically coupled co-simulations with it.

Based on the findings of the literature study and the requirements of the thesis, the data models and communication patterns will be designed. The data model will consist of new data types and their associations based on the defined requirements with the specified data transfer protocols in mind. The application level communication pattern for the co-simulation system will be designed around SOAP and OPC UA as those two protocols are the first protocols to be implemented for this thesis.

After the design phase, a proof-of-concept implementation of the co-simulation system will be created. The system will be able to execute a co-simulation between the Matlab/Simulink and the OpenModelica simulation tools, using SOAP and OPC UA as the respective protocols for the data exchange between them.

The last step will be the testing and evaluation of the different used simulation tools and protocols based on their speed, their usability and their extendability. An overview will be given to highlight advantages and drawbacks of the different technologies based on the aforementioned criteria.

CHAPTER 2

# State of the Art

## 2.1 Co-Simulation

A lot of scientific fields today are supported by computational models and simulation. Models are abstract representations of the real world. They are formulated through mathematical relationships and describe how a system behaves and changes over time. Models are simulated with the help of simulation tools to gain insights into the behaviour of complex systems. Simulation itself is the act of numerically calculating solutions to the model's equations over a specified time period. A distinction can be made between continuous and discrete models/simulations [BB14].

Continous models are based on mathematical equations and describe continous processes in the real world, like electrical systems or heat distribution. Such models typically contain ordinary and/or partial differential equations (ODE and PDE) and are simulated with the help of ODE/PDE solvers, optimized algorithms to numerically solve these systems of equations [CK06].

Fundamentally different to continuous simulations are discrete event simulations. They do not simulate continuous processes, but instead are based on events at specific time instants. A solver for discrete event simulation does not need to continously solve the system, but only needs to do so at the time instants at which an event occurs. To find/compute these time instants is a big challenge when designing a discrete event simulation software [CK13].

The term co-simulation describes the simulation of a system in a distributed manner. The parts of a co-simulation do not necessarily have to be of the same type. The problem that this approach also solves is that continuous and discrete models are hard to combine/simulate in one single simulation environment. In a co-simulation, the continuous model can be simulated with a tool for continuous simulation and the discrete

part can be simulated with a discrete simulation tool. The data that each sub-model needs from other sub-models is exchanged between simulation runs.

To be able to distribute the model, the system is split into sub-parts, where each part acts as a black box with inputs and outputs that can be simulated indepenently from the other parts. Inputs for a part of the co-simulation are the values it needs from other parts to simulate its own sub-model. The outputs of a part are the values it provides for other sub-models. Parts of a co-simulation typically exchange these values in defined intervals. The duration of an interval therefore directly affects the precision of the co-simulation. If the intervals are short, the co-simulation will take longer to execute but the results will be more precise because the shared variables will be updated more often. Likewise, if the intervals are long, the co-simulation will execute faster but the results will be less precise, depending on the influence of the shared variables on the sub-models. Three general approaches to *coupling* a co-simulation exist to deal with this problem.

### Weak Coupling

*Weak* coupling is the most basic form of connecting the parts of a co-simulation, but is nonetheless often used due to its speed and simplicity compared to the other two approaches. In this form of coupling, each part of the co-simulation simulates its part of the model. When the parts are finished with simulating for some set interval, they exchange data and simulate the next interval. This is done until the co-simulation is finished. The precision of the results in this form of coupling is only influenced by the amount of time between data exchanges, where a longer time between exchanges generally means less accuracy. The loss of accuracy stems from the fact that the simulation tools do not have any information on the value of the datapoints of other simulations between exchanges. Therefore they can only be extrapolated by the simulation tool in some fashion. If more time passes between such an exchange, the greater the difference between the used value and the actual value of the datapoint can become. Errors emerging due to this inaccuracy can propagate through the whole co-simulation and falsify the results. This problem can be somewhat mitigated by using more sophisticated coupling strategies where values for the datapoints are interpolated between the exchanges. An in-depth mathematical analysis of simulation and different co-simulation strategies can be found in [Awa15] and [Sch15]. Figure 2.1 shows how a co-simulation is executed when using *weak* coupling.

### Parallel Dynamic Coupling

*Parallel dynamic* coupling is a more sophisticated form of co-simulation where each part of the co-simulation can interpolate the values of other parts. Using this coupling strategy involves simulating one interval multiple times. Figure 2.2 illustrates the sequence of events in this coupling mode. In the first iteration in each step of the co-simulation, each part of the co-simulation has the results of the other parts at the last iteration of the

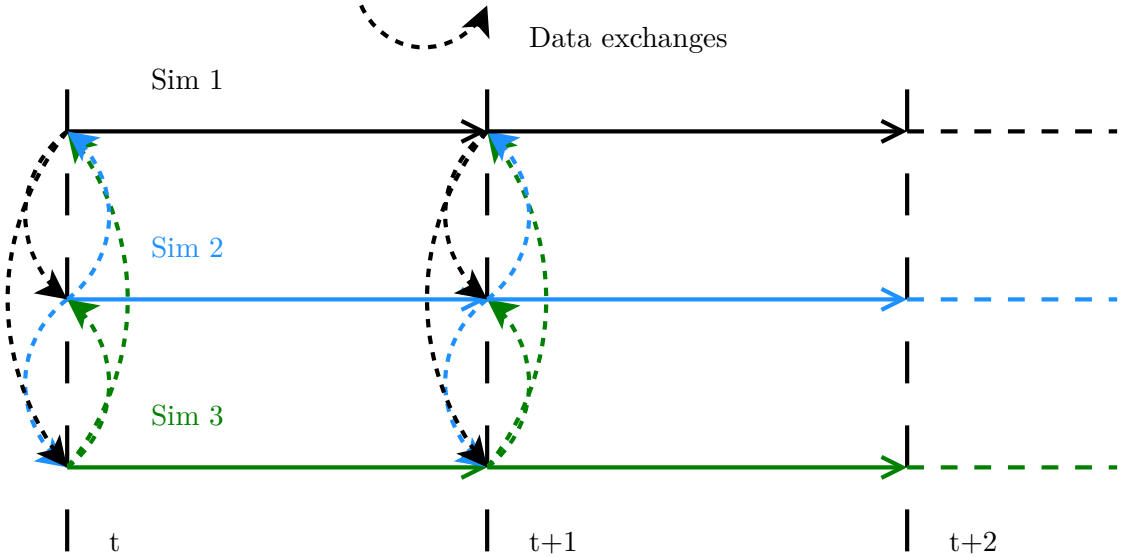


Figure 2.1: Sequence of events in weak coupling mode

last interval. This is one value per variable. Now each part simulates its model for the duration of the interval and then the parts exchange data again. Each part now has two values for each variable from the other parts. The first value is the value of the variable at the start of the interval and the second value is the value of the variable at the end of the interval. Now the parts reset back to the beginning of the interval. With this knowledge of the future, a part in the co-simulation can interpolate the value in its own simulation from the start until the end of the interval. This process can be executed multiple times per interval to potentially achieve greater accuracy [AG01][Whi+85].

### Serial Dynamic Coupling

*Serial dynamic* coupling is a variation of dynamic coupling where the parts of the co-simulation execute after one another. The basic principle is the same as in *parallel dynamic* coupling, but the amount of iterations needed for the same precision are potentially far less. Figure 2.3 shows the sequence of events in the serial coupling mode. Parts 1, 2 and 3 exchange data at the start of the interval, then part 1 starts with the simulation of the interval. After it has finished, it sends model parts 2 and 3 its data from the end of the interval. Part 2 now starts its first iteration of the interval and is already able to interpolate the values of part 1. After part 2 has finished, part 3 can start to simulate and has interpolation values for both, part 1 and 2. The last part to execute the iteration is able to interpolate all variables from other parts of the co-simulation and will produce the most accurate results. This process can also be repeated multiple times per interval to achieve greater accuracy.

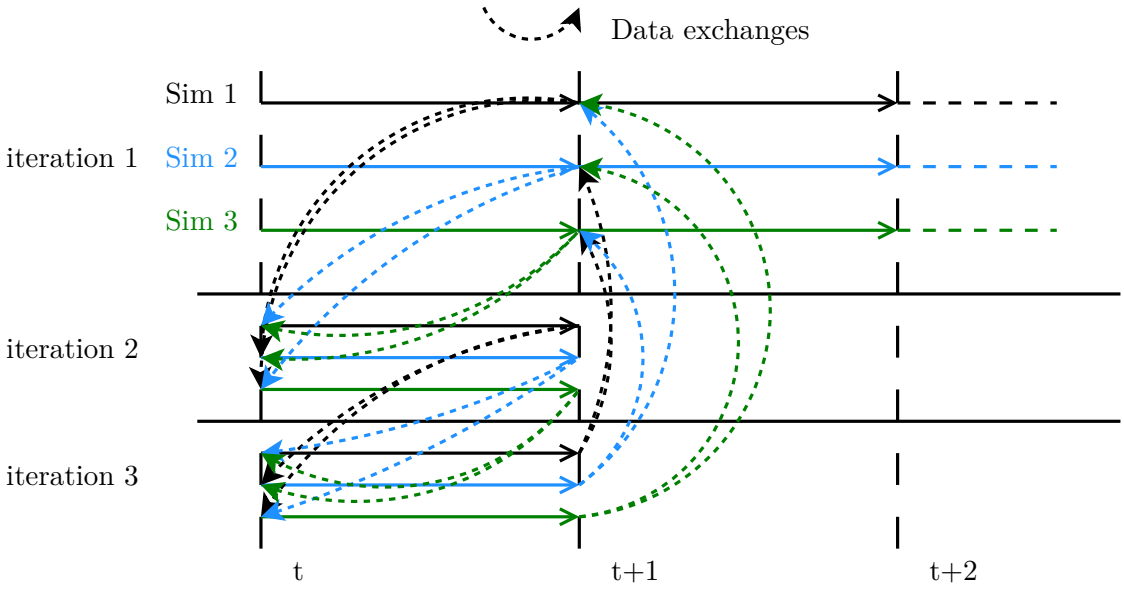


Figure 2.2: Sequence of events in parallel dynamic coupling mode

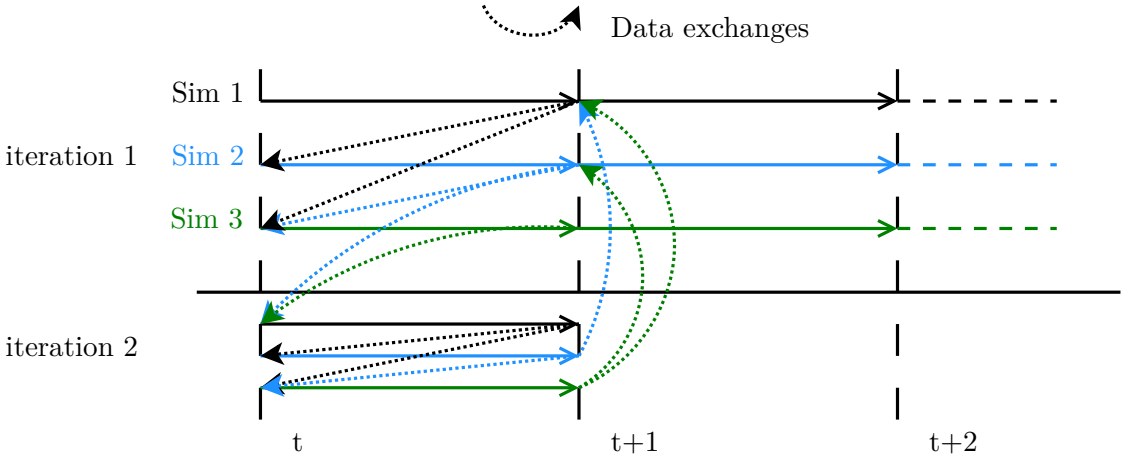


Figure 2.3: Sequence of events in serial dynamic coupling mode

## 2.2 Latest Developments

Due to the current developments in the area of Cyber-Physical Systems, state-of-the-art research in this field is largely based on co-simulation between continuous and discrete simulations, but most of this research focuses heavily on specific simulation tools.

[Lin+11] describe a co-simulation framework which takes both, the regular intervals of the continuous and the events of the discrete simulation into account and forms a global timeline with information on when information is needed by which simulation. Based on this timeline, the simulations can be synchronized without loss of accuracy. A drawback of this approach is that all discrete event times have to be known in advance.

Research in different coupling algorithms is done by Ciraci et al. [Cir+14a][Cir+14b]. In their research, they present a framework for co-simulation which uses optimistic approaches to predict when the next synchronization should take place. Also, various other approaches for coupling co-simulations are presented, those can be used to reduce the overhead and the number of the synchronizations between simulations. The approaches yield varying results depending on the involved simulations.

Often, the co-simulation is implemented only locally by using named pipes or other inter-process communication methods [LA11][Ton10][Lin+11]. This of course reduces communication overhead and speeds up the overall co-simulation between two or more specialized simulation tools, but reduces interoperability and versatility. Additionally, it is often hard to extend models which are based on those co-simulation systems with new variables or introduce new models into the co-simulation setup.

Other implementations like BCVTB [Wet12] allow for a variety of simulation environments to be used and coupled together. BCVTB works by using a middleware which controls data flow between the individual simulators. Data between the actors is exchanged as raw data via BSD sockets, without metadata like units or names. BCVTB is well suited for fast co-simulation involving different simulators but the way in which data is exchanged makes it hard to extend existing models with new datapoints and add new models altogether.

A different approach for co-simulation is the Functional Mockup Interface [Blo+11]. Individual simulation units can be compiled before the actual simulation with their own solver in place. The FMI provides an interface which the units have to implement for them to be used later in a co-simulation setup. At simulation time, the individual simulation units, which are run by their individual simulation tools, are handled by a central master that uses the FMI interface to coordinate the units. With the help of wrappers, communication between the master and the individual units can be extended to function over a network.

In 1998, an approach was made by the US Defense Modeling and Simulation Office to create a standard (High Level Architecture) for distributed modelling and simulation [Boa10]. This standard has been revised and extended in 2010 to include various

web service technologies, among which is the support for WSDL. HLA is a solid standard for the federation of simulations, but it is not an open standard and is mainly used in commercial and governmental environments.

### 2.2.1 Advantages of Existing Approaches

The advantages of existing co-simulation implementations such as [LA11] are the speed at which they execute the co-simulation. Because of the tight integration of the co-simulation framework into the simulation environment and the low level data exchange (shared memory/low-level socket data transfer), the data can be exchanged much faster. Also, the integration of the co-simulation functionality into the simulation tool itself enables the co-simulation to use the event-detection mechanisms of the simulation tools and thus be more accurate than other approaches.

### 2.2.2 Drawbacks of Existing Approaches

The drawbacks of tightly integrated co-simulation frameworks are usually that they are not scalable and do not have a high degree of extendability or usability. Often the framework is tailormade for two specific simulation tools or even a model and so can not be reused easily.

Most implementations separate the simulation into a discrete and a continuous part, and usually both parts are run on the same machine. In these scenarios, co-simulation is used for accuracy in systems where only one continous and one discrete simulation is involved. But with the growing amount of data in simulations it would be preferrable to be able to split a simulation into multiple parts and execute them in a distributed environment, maybe even in the cloud.

## 2.3 Modelling and Simulation Tools

### 2.3.1 Matlab and Simulink

Matlab is a tool for numerical computation of various problems, it was created and is still developed by MathWorks [Mat]. Built-in is a versatile scripting language that can be used to create classes and functions. A very important feature for this thesis is the support for an easy creation of a SOAP client from a WSDL file.

Simulink is a graphical tool for designing models of continuous systems. It is created by the same company that develops Matlab, and is integrated into it. Simulink offers some ways to interact with Matlab at the start/end and even during a simulation, which makes those two tools combined an ideal candidate for one part of the proposed co-simulation tool.

### 2.3.2 OpenModelica

OpenModelica [Cona] is a modelling environment based on the Modelica [Ass] language. It is free of charge and is actively maintained and developed. The Modelica language is an object-oriented declarative modelling language designed for continuous simulation. Model binaries created with the OpenModelica compiler can load initial values for all components from a file given at the start and export the values of all components of the model to a file at the end of the simulation. This capability made OpenModelica a prime candidate for use in this thesis.

## 2.4 Communication Protocols

### 2.4.1 SOAP

SOAP, or originally Simple Object Access Protocol, is a protocol for exchanging structured data in the form of XML. Early versions of SOAP were developed in the late nineties for Microsoft. It was not until 2003 and SOAP version 1.2 that it became a World Wide Web Consortium (W3C) [Conb] recommendation, and thus a "standard".

SOAP defines the basic XML structure of the content in messages. Messages are contained within an `<Envelope>` tag and consist of optional `<Header>` elements and a mandatory `<Body>` element. The header elements can contain arbitrary meta data for the message, while the body element contains the message content. Child elements of the `<Header>` and `<Body>` elements are defined by the application itself via namespaces.

The SOAP standard basically defines a protocol between the transport and the application protocol. This means that the application itself has to process the SOAP data but many libraries and tools exist which can handle this task.

Listing 2.1 shows an example message based on the data used in this thesis.

Listing 2.1: SOAP structure example

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <registerResultRequest xmlns="http://www.pueh.at/">
      <stateId>54</stateId>
      <iteration>1</iteration>
      <simulationResult>
        <simulationName>thermisch</simulationName>
        <dataset>
          <time>2650.0005</time>
          <datapoints>
            <name>TZ1_Pel_O</name>
```

```
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
            <valueArray>
              <value>500.0</value>
            </valueArray>
          </datapoints>
          <datapoints>
            ...
          </datapoints>
          ...
        </dataset>
      </simulationResult>
    </registerResultRequest>
  </S:Body>
</S:Envelope>
```

Because SOAP only defines the structure of messages, it relies on other protocols, such as HTTP, for the transportation of the messages to an endpoint. The Web Service Description Language (WSDL) is often used to easily create SOAP web services. WSDL is an iterface definition language, where the endpoints of a service are defined and described so that clients can automatically access them. The definition tells clients which operations are supported and how the data is structured in the request and the response. This allows for the automatic creation of clients by tools like the Apache CXF framework [Foua]. The use of these open and widely used standards makes SOAP platform independent and ideal for cross-platform applications. Because Matlab offers integrated support for the creation of SOAP clients from WSDL files, SOAP was the obvious choice for a data-exchange protocol between Matlab and the central server in this thesis.

A disadvantage of using SOAP is that, by default, it uses plain-text XML to transfer data. This leads to a high overhead in transmitting and parsing the data due to the text-heavy nature of XML. However, other transmission mechanisms and formats, like the Message Transmission Optimization Mechanism [Groa], which uses XML-binay Optimized Packaging [Grob], exist to alleviate this issue.

### 2.4.2 OPC UA

The OPC Unified Architecture standard is the successor to the OPC standard, which is widely used in industrial applications for exchanging data. This new standard was and is still developed by the OPC Foundation [Fouc]. The standard is platform-independent and based on a service-oriented architecture to further enhance interoperability between

different systems. The two main features of OPC UA, upon which all others are built, are the transport protocols and the information meta-model.

The information model defines how datatypes are constructed. OPC UA defines some basic data types and an inheritance mechanism that can be used to build custom data types. Data types can contain simple fields, functions and references to other types. Like in object-oriented programming languages, types can be instantiated as concrete objects of that type. Types, and their instances are linked together via hierarchial and non-hierarchial references. In addition to hierarchial references, items can have non-hierarchial references to one another so as to be able to associate them in different ways. An actuator in one branch of the model might, for example, be dependent on the value of a sensor in another part of the model without a parent-child association. The ability to create custom type information is very useful when defining the semantics of a custom type. Nodes can for example be typed as temperature sensors or stepper motors. Another very useful and related feature is the built-in ability to give values a unit. A node with type heat-sensor, for example, can have a datapoint of type float with unit °C. This furthers interoperability and automation between different systems.

Other features of OPC UA include the ability to store historical data, an alarm system, support for redundancy and heartbeats and many more. Of great importance are also the built-in security mechanisms with support for authentication, authorization, encryption and data integrity via signatures. Fast and secure communication between low-power peripherals is important in industrial as well as private environments, especially with the proliferation of the Internet of Things.

Figure 2.4 shows the different protocols that can be chosen for exchaning data. The most interesting one for this thesis was the *opc.tcp* protocol. This protocol is built on top of the TCP protocol and offers high speed and low data usage when transferring data over the network. The other available protocol option would be HTTP where the server is used as a web service and the data is transferred via SOAP. Whereas the binary protocol is implemented in all available OPC UA libraries, the web service is currently only available in some of them (e.g. .NET).

In this thesis, the OPC UA binary protocol is used for data exchange between the central server and OpenModelica simulations, where the OpenModelica simulation runs the OPC UA server and the central server queries the simulation for the needed data.

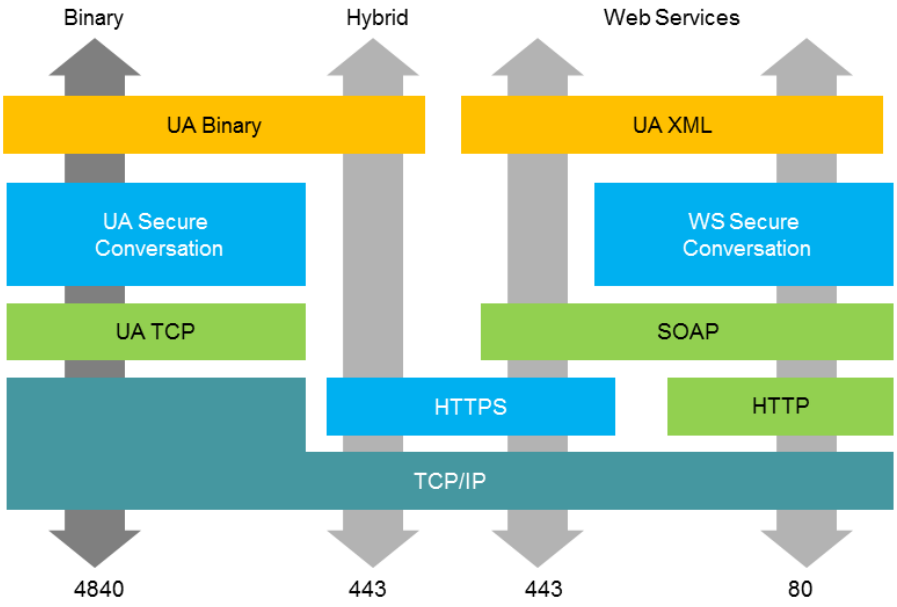


Figure 2.4: Different data transfer modes in OPC UA [Aro]

CHAPTER 3

# Requirements

The central goal of this thesis is the implementation and analysis of a co-simulation tool which enables distributed co-simulation of Simulink and OpenModelica models. Below is a list of functional and non-functional requirements.

### Functional Requirements

1. **Support for weak coupling.**

2. **Support for dynamic parallel coupling.**

3. **Support for dynamic serial coupling.**

4. **Support for engineering units as metadata.** The communication protocol will include the engineering units of the sent datapoints as metadata. This reduces errors due to wrong assumptions about units and can be used to automatically convert between them. Additionally, this serves as a demonstration as to which benefits can be gained by sending additional metadata.

5. **A central server for data exchange.** It should be possible for multiple simulations to participate in a co-simulation. Therefore, a server will be implemented that communicates with each of the simulations and is responsible for exchanging data between them. This server will have the following capabilities:

    a) *Support multiple break conditions for dynamic coupling.* In order to limit the amount of iterations for dynamic coupling, the co-simulation should automatically advance to the next interval if a maximum number of iterations is reached or if the error for all datapoints has become small enough.

b) *Support an ordering of simulations for dynamic serial coupling.* It should be possible to set an ordering for each participating simulation, which specifies at which point in an iteration it will be executed.

c) *Set initialization data.* The server will be able to set the simulation parameters of each individual participating simulation based on the configuration of the server/the co-simulation. These initialization parameters include: the used solver, the start time of the simulation, the end time of the simulation as well as the interval between data exchanges.

d) *Convert between units.* The server will be able to convert between different units of datapoints, based on what unit a simulation sends and what unit a simulation expects. This requirement is restricted to converting between different powers of the same base unit, but the mechanism should be easily extendable to include different conversion mechanisms.

e) *Support SOAP data transfer.* In order for simulations to communicate with the server via SOAP, the server will contain an HTTP SOAP server which receives requests from the simulations (SOAP clients).

f) *Support OPC UA data transfer.* The server will have the means to communicate with simulations via OPC UA.

6. **A co-simulation framework for Matlab/Simulink.** It should be possible for Simulink simulations to participate in a co-simulation. Therefore, a framework will be developed in the Matlab scripting language, which will have the following features:

   a) *SOAP client for communicating with the server.* In order to communicate with the central server, the Matlab framework will use Matlab's built-in functionality to automatically create a SOAP client from a WSDL file.

   b) *Ability to execute Simulink simulations.*

7. **A co-simulation framework for OpenModelica** It should be possible for OpenModelica simulations to participate in a co-simulation. Therefore, a framework will be developed which will have the following features:

   a) *Ability to communicate via OPC UA.* In order to communicate with the central server, the framework will be able to communicate via OPC UA.

   b) *Ability to execute OpenModelica simulations.*

### Non-functional Requirements

1. **Usability.** The co-simulation tool should be as easy to use as possible.

   a) *Low amount of modifcations for an existing model.* The amount of changes that are needed for a model to be used in a co-simulation should be kept to a minimum.

b) *Easy integration of a new simulation into a co-simulation.* It should be easy to add a new simulation to the co-simulation setup.

c) *Easy configuration of server and frameworks.* The amount of configuration that is needed for the server and the frameworks should be kept to a minimum.

2. **Extendablity.**

   a) *More simulation tools.* It should be easy to create a new framework for a new simulation tool and integrate it into the existing setup.

   b) *Diversity of data transfer protocols.* It should be easy to extend the central server with more communication protocols, so that a new framework can be integrated without too much effort.

   c) *Support of further unit conversion algorithms.* It should be easy to implement new strategies to convert between units.

3. **Scalability.** It should be possible to have an unlimited number of simulations participating in one co-simulation, with each simulation running in a pure distributed fashion (e.g. on individual hardware).

CHAPTER 4

# Design & Implementation

## 4.1 Overview

Based on requirement 5, the communication between the involved simulations happens via a central server, the so-called *Coordinator*, which handles all the data exchange and the flow of the co-simulation. The Coordinator uses two protocols for communicating with the simulations, XML-based SOAP, and OPC UA.

Figure 4.1 shows an overview of what the complete system looks like and how the components are interacting. Data is repeatedly transferred from the simulations to the Coordinator, where it is processed and sent to the other simulations so they can use it.

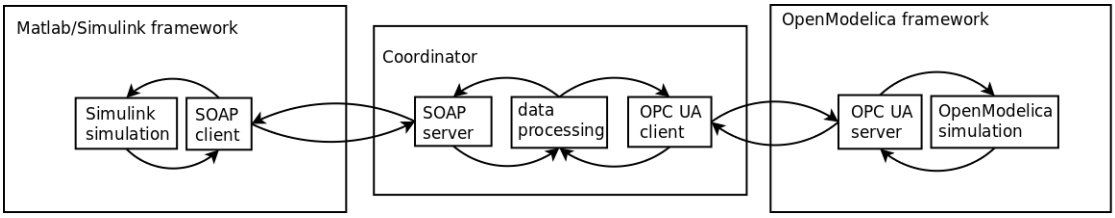


Figure 4.1: Overview of the components and their interactions

## 4.2 Chosen Technologies

Due to its speed, C++ was chosen as the programming language for the Coordinator. It is the central piece of the setup and has to be fast or it will be the bottleneck of the co-simulation. In order to utilize new features and be up-to-date with technology, the C++11 standard is used for compilation and development.

A co-simulation setup can contain both Matlab and OpenModelica simulations. Because of this, the Coordinator contains a SOAP server and an OPC UA client in order to communicate with the two types of simulations. In this setup, it can directly initiate communication with the OPC UA-based OpenModelica framework, whereas it has to passively wait for data to be sent from the Matlab framework via SOAP. There are positive and negative aspects to both approaches, those are explained in more detail below.

The Matlab/Simulink framework (requirement 6) is developed in the Matlab scripting language. This framework handles the running of Simulink simulations, the extraction of results from the simulations and the communication with the Coordinator via a SOAP client. Because Matlab can create SOAP clients nearly out-of-the-box from a WSDL file, it was decided that the framework should act as the client and the Coordinator as the server. This setup is somewhat unintuitive because the Coordinator, as the manager of the co-simulation, has to wait for the frameworks to contact him, instead of him remotely calling the frameworks.

The OpenModelica framework (requirement 7) is also developed in C++ and handles the running of simulations and extraction of results from OpenModelica simulations. It communicates with the coordinator via OPC UA. To this end, the framework contains an OPC UA server which provides all the information about the simulation as well as the results from a simulation step. In this case, the Coordinator acts as the client. When the Coordinator can directly send and retrieve data from the frameworks, as well as call functions remotely on the frameworks, synchronization and error handling becomes a lot easier.

To have two different communication models is somewhat counter-intuitive. The reasoning behind this is that Matlab supports SOAP clients out-of-the-box, therefore it was much easier to implement it. On the other hand, when the Coordinator acts as the initiator of requests (OPC UA), communication is much more intuitive and easier to control. In hindsight, having both communication patterns gives a good overview of the advantages and drawbacks of each approach.

**Coordinator acts as server:**

**Advantages**

- Only the Coordinator address information needs to be known by the frameworks.
- NAT mechanisms only have to be taken care of on the Coordinator side.

**Drawbacks**

- Coordinator has no way to initiate communication with the frameworks.
- Flow of control is unintuitive, framworks initiate communication, Coordinator has to wait for the information.

- Coordinator has no real way of knowing if a framework has crashed, only via timeout.

**Coordinator acts as client:**

**Advantages**

- Coordinator can directly initiate communication with the frameworks.
- Flow of control is intuitive, Coordinator controls frameworks.
- Coordinator can immediately tell if a framework is not reachable.

**Drawbacks**

- Coordinator needs to know address information for each of the frameworks.
- NAT mechanisms have to be taken care of for each framework.

SOAP was chosen because of its extendability and descriptiveness in terms of transferred information. SOAP will likely lack somewhat in speed because of its communication model (server, client, new request for each data transfer) and the amount of metadata transferred. OPC UA will probably be significantly faster than SOAP but is much more complex, and thus not quite as flexible as SOAP.

## 4.3 Data Model

Figure 4.2 shows an overview of the relevant datatypes and their associations within the Coordinator. Each technology (SOAP and OPC UA) has its own way to encapsulate and transfer the data. The Coordinator converts the received data to its internal representation. Similarly, when sending data to the simulations, the Coordinator converts the internal representation to the representation that the technology uses. By using this internal representation, the core of the Coordinator is independent of the used technologies and can work on the data without considering the transfer mechanisms.

Data is sent between the actors in the co-simulation setup as “simulation results”. A simulation result contains the name of the simulation which produced the data and a dataset containing a timestamp and a list of datapoints with values and units. The timestamp of the dataset indicates when the data was produced. A unit contained in a datapoint contains a string representation of the unit name and a power, which indicates the offset from the non-prefixed unit in terms of $10^{power}$. An example would be a unit `[name = "kW", power = 3]`, which means that the unit has to be multiplied by $10^3$ to get a value without prefix. This comes into play in the Coordinator, which

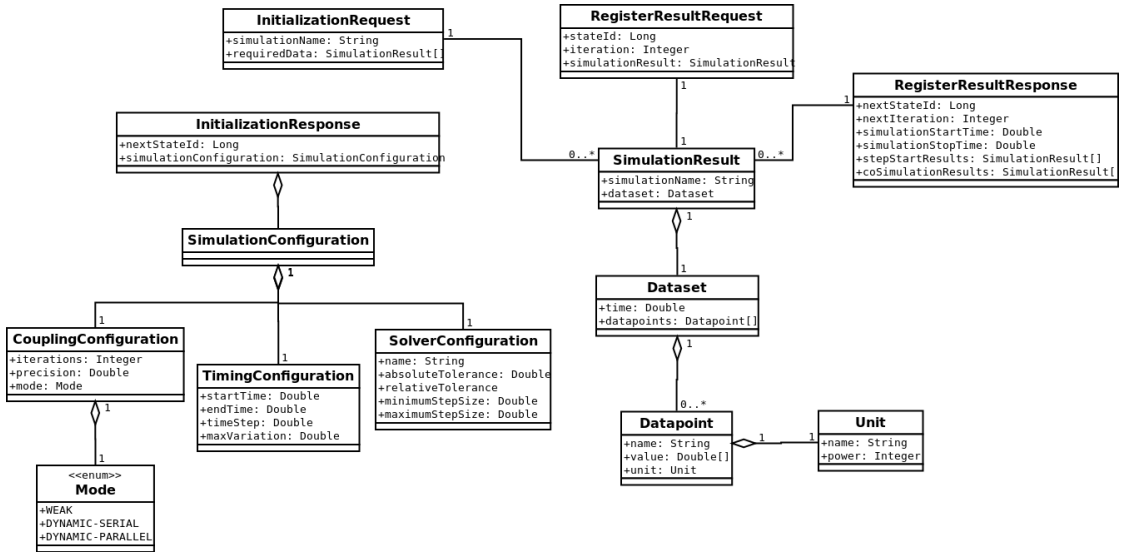


Figure 4.2: Overview of datatypes and associations

automatically converts values based on the unit sent by a simulation and the units which are required by other simulations in the setup. Currently, each datapoint is designed to hold a vector of values. This feature is not used in the current implementation but can potentially be used in later versions to transmit a vector as one single datapoint instead of individual ones.

Initialization request data, where a simulation requests certain datapoints from other simulations, is also sent in the form of simulation results, but the dataset timestamp and the datapoint values are omitted in this case because they hold no significance. Figure 4.4 shows how such an initialization request message looks like.

Each response message from the Coordinator contains a `nextStateId` field which indicates the state that the Coordinator is in and which message it expects from a simulation. The next message from the simulation to the Coordinator has to have its `stateId` field set to the last `nextStateId` it received from the Coordinator. This keeps the simulations and the Coordinator synchronized and helps to detect errors.

Initialization response messages from the Coordinator to the simulations also contain configuration data for the co-simulation. This data is split into three parts:

1. **Coupling configuration**, contains the coupling mode (either `WEAK`, `DYNAMIC-PARALLEL` or `DYNAMIC-SERIAL`), the number of iterations for each interval (only important in dynamic coupling), and a precision. The precision can be used in dynamic coupling to proceed to the next interval once the difference of two values of the same datapoint in successive iterations is less or equal to the precision, see Section 4.3.1 for details.

2. **Timing configuration**, contains the start and end time of the whole simulation, the interval (the time for one simulation cycle) and a maximum variation, which indicates the maximum time difference between expected and actual time in simulation results which the Coordinator tolerates before terminating the co-simulation.

3. **Solver configuration**, contains control data for the solver, such as the name of the solver to be used, absolute and relative tolerance, and minimum and maximum step size.

### 4.3.1 Data for Dynamic Coupling

Some parts of the data model are specifically used for dynamic coupling and are meaningless in weak coupling mode. Those parts are:

1. **The number of iterations**: Indicates how many iterations of the same interval should be simulated (at most).

2. **The precision**: Defines a termination condition in terms of a limit. If a precision is specified, the co-simulation advances to the next interval of the simulation, if

$$|dp_{j,i} - dp_{j,i-1}| < precision$$

holds true for all datapoints $dp_j$ in each iteration $i$, where $j$ is the index of the datapoint in the co-simulation. If a number of iterations and a precision are defined, the co-simulation will advance if the precision criteria is met or if the maximum number of iterations has been simulated. If no number of iterations is defined, an interval will be iterated until the precision criteria is met.

3. **The step-start-results**: In dynamic coupling mode, the Coordinator additionally sends step-start-results at iteration 1 of each new interval. The step-start-results contain the results of the simulations at the end of the last interval. In contrast, the co-simulation results are updated after each iteration with the current results of the simulations. For example, take a simulation run `[iteration=1, startTime=1, stopTime=2]`, where the results of `SimulationA` are `[a=2.5]` at time 1. Other simulations will receive `[stepStartResults=["SimulationA",a=2.5], coSimulationResults=["SimulationA",a=2.5]]` at the start of the simulation run. After this run, the results of `SimulationA` are `[a=5.5]` at time 2. The data sent to the simulations for the next simulation run `[iteration=2, startTime=1, stopTime=2]` will be `[coSimulationResults= ["SimulationA",a=5.5]]`. Now, the other simulations can interpolate values for `SimulationA.a` by using the value in the step-start-results and the current value in the co-simulation-results. For parallel dynamic coupling, no extra field like this would be necessary, but serial dynamic coupling necessitates this because in this mode, step-start-results and co-simulation-results can already be different

in the first iteration. Suppose `SimulationB` is the second simulation in a serial simulation run, and `SimulationA` is the first. The results of `SimulationA` in the first interval `[startTime=0, stopTime=1]` are `SimulationA.a=2.35`. Now interval `[startTime=1, stopTime=2]` starts and `SimulationA` simulates first, for `SimulationA`, the step-start-results and the co-simulation-results will be identical. After this first simulation run, the value of the datapoint of `SimulationA` has changed to `SimulationA.a=5.12`. All simulations that run after `SimulationA` will receive different values for the step-start-results and the co-simulation-results: `[stepStartResults=["SimulationA","a"=2.35], coSimulationResults=["SimulationA","a"=5.12]]`. This way, later simulations in the order can already interpolate in the first iteration.

## 4.4 Communication Model

At the beginning of each co-simulation, all involved simulations have to send initialization data to the Coordinator, which contains datapoints that the simulation needs from other simulations in the co-simulation setup. In the case of OPC UA, the Coordinator initiates the connections to the simulations and polls the data directly. The Coordinator then sends initialization data for the simulations. The initialization response contains configuration data for the solver, the timing, and the coupling mode of the co-simulation. After this initialization step, the simulations begin simulating their model and their results are regularly transmitted to the Coordinator. The Coordinator then sends some control data and the datapoint values of the other simulations in the setup. See Section 4.3 for details on the data model. An overview of the communication process can be seen in Figure 4.3. The different actors are all part of the Coordinator. In the case of a parallel co-simulation, the Coordinator instantiates a ParallelSimulator object which handles the flow of data. In a parallel co-simulation, each participating simulation executes an iteration simultaneously. If the client simulation framework uses SOAP, the SoapServer waits for connections with the framework and then calls back to the ParallelSimulator, which processes the data. If the client framework uses OPC UA, the OpcUaSimulationConnector directly initiates communication with the OPC UA server and calls back to the ParallelSimulator when the data has been fetched. The calls back to the ParallelSimulator block until data from each client is available and the next iteration can be simulated. Figure 4.4 shows how the initialization messages are structured.

### 4.4.1 SOAP Communication

Communication between the Coordinator and the Matlab framework happens via SOAP. The datatypes and services are described in a WSDL file. The WSDL description contains a binding for a SOAP service which is used by the SOAP server contained in the Coordinator, and the SOAP client contained in the Matlab framework to structure

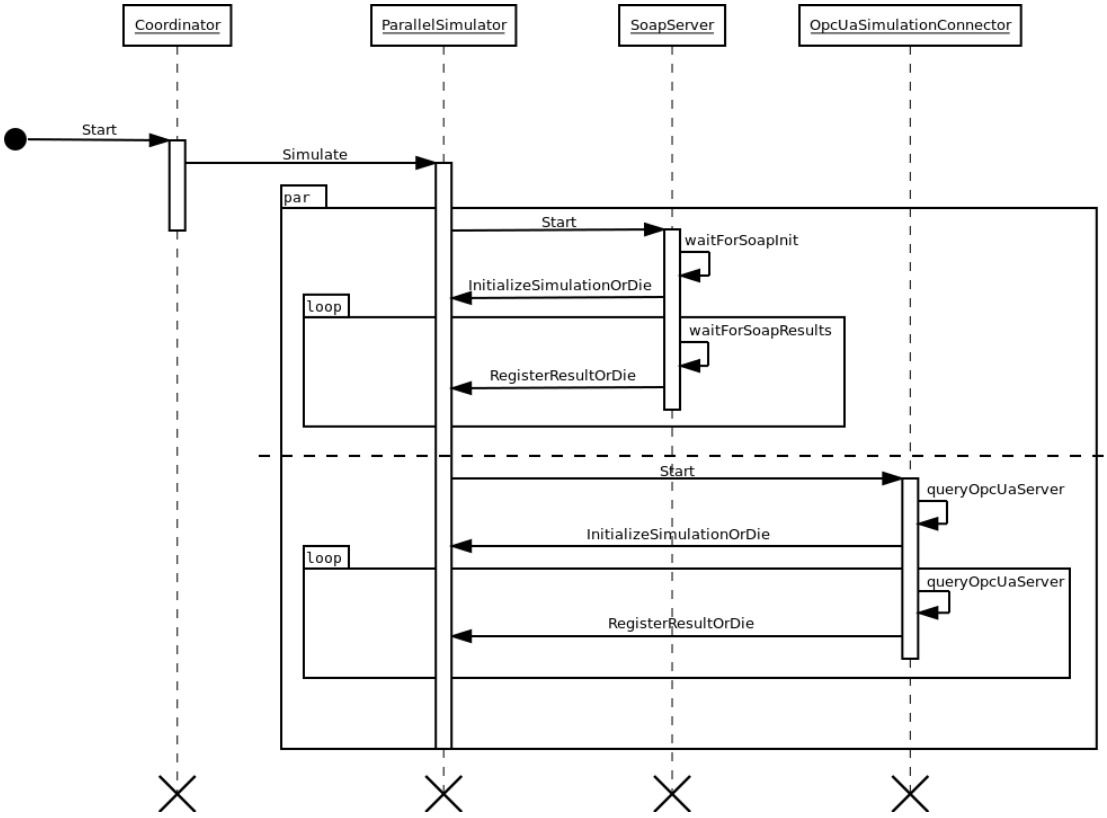


Figure 4.3: Basic communication model for parallel simulation

their data accordingly. The datatypes used in this SOAP exchange are essentialy the same as described in Section 4.3. Because of the client/server nature of HTTP transfer, it originally made sense that the Coordinator should be the HTTP server where each SOAP-based simulation would register their results. One big reason for its current design is that Matlab has no functionality to create a SOAP server, but it has built-in support to automatically create a SOAP client from a WSDL file. Another reason for this control-flow direction is that the Coordinator does not need to know the IP-addresses and ports of each simulation at startup, it just waits for the simulations to connect to its server.

### 4.4.2 OPC UA Communication

OPC UA communication happens in the opposite direction of the SOAP communication. Here, the Coordinator contains an OPC UA client and the simulation framework contains an OPC UA server. Each server of a simulation has two methods, `initialize` and `simulate`, which are called synchronously by the Coordinator. Before each call to `simulate`, the Coordinator writes the results from the other co-simulations to the

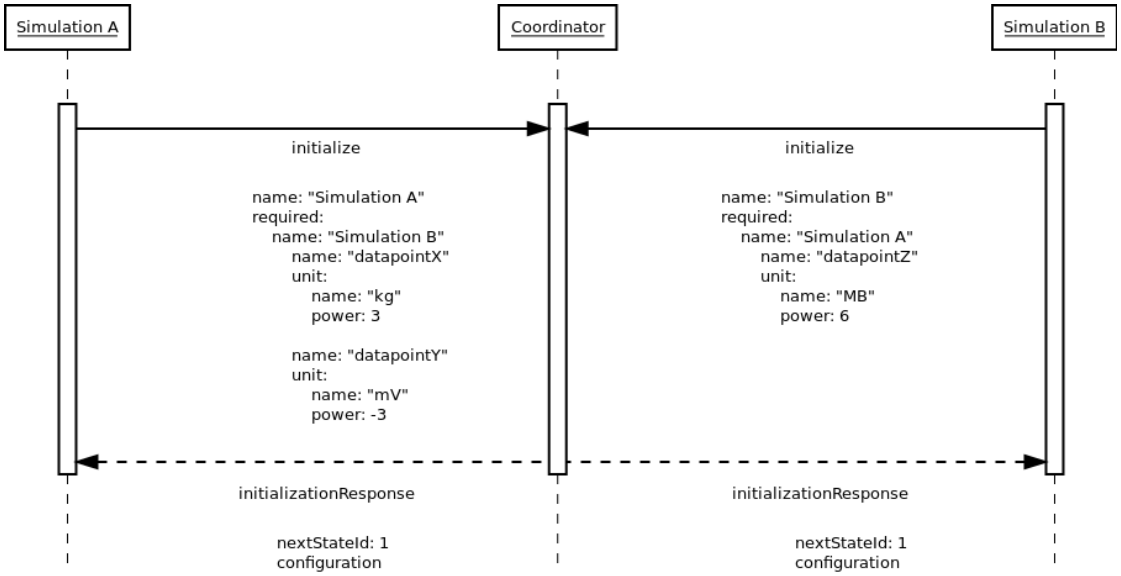


Figure 4.4: Initialization messages from the simulations

server and after each call to `simulate`, it reads the results of the simulation from the server. OPC UA binary encoding is used as the transfer encoding, as it offers very good performance in terms of bandwidth and speed.

## 4.5 The Coordinator

The Coordinator acts as the central data exchange and coordinates the pacing and type of the co-simulations via control data sent to the individual simulations. The basic layout of the Coordinator can be seen in Figure 4.5.

The Coordinator connects to the simulations via so-called *SimulationConnectors* that provide data from and send data to their respective simulations. The SimulationConnectors are instantiated at startup based on the configuration given to a *Simulator*, which can be either a Serial- or ParallelSimulator. The *Simulator* controls the pacing of the whole co-simulation, it tells the SimulationConnectors when to proceed with which simulation step. A ParallelSimulator waits for every simulation to register a result after each step, and then tells all SimulationConnectors to proceed simultaneously. A SerialSimulator lets the SimulationConnectors execute in series, providing the following simulations with results of the previous simulations from the current interval. This approach is only useful when using dynamic coupling and should in theory provide more precise results in less iterations than a parallel approach. The state of the co-simulation is managed by the CoSimulation object, which is also instantiated based on the type of co-simulation. It saves a history of all the datapoints sent to the coordinator over the whole co-simulation. This history can be saved as a Matlab or XML file for later use.

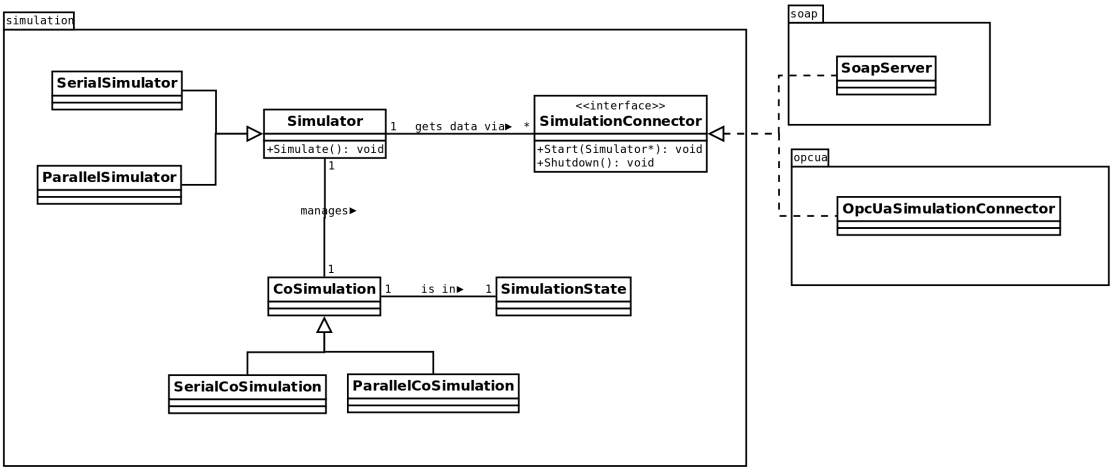


Figure 4.5: Basic class layout of the Coordinator

### Building and Starting the Coordinator

The Coordinator is implemented in C++ in order to get the most speed out of this central piece in the co-simulation setup. It is developed for the C++11 standard so that new convenient features could be used when implementing it. For building the Coordinator, `cmake` [CMa] is used. This tool offers a great deal of automation and flexibility and is a well-established cross-platform buildsystem generator. Several external libraries are used to get the intended functionality for the Coordinator. To run the SOAP server, the gSOAP [Eng] library is used. This library, in conjunction with the included binaries, offers very easy integration and usage of a SOAP server. The gSOAP library is published under the GPL v2 license and thus can be used freely for the purpose of this project. To run the OPC UA client, the Unified Automation [Aut] SDK was used. This SDK contains an "evaluation licence" which makes it possible to use the functionality of the SDK for up to one hour before it stops working. This also means that co-simulations with OPC UA participants can run at most one hour before the co-simulation fails. In order for the OPC UA SDK to compile, the additional libraries *xmlparser*, *xml2*, *ssl* and *crypto* are needed. *TinyXML2* [Tho] is used in order to parse XML files for initializing the Coordinator. It is a library under the zlib license and may be freely used for any purpose. Lastly, *Log4cxx*, published under the Apache v2 License is used for logging purposes.

### The Build Process

The steps described here are for understanding only, they are automatically executed when `make` is run in the coordinator directory. All the commands and steps are integrated and executed via the *CMakeLists.txt* file. For convenience, a custom `makefile` was created to automatically build the project out-of-source in the `build` directory.

Before compiling any code, the sources for the SOAP server have to be generated via the binaries delivered with the gSOAP library. First, `wsdl2h` is executed, which takes as input the WSDL file `../cosimulation.wsdl` and outputs the header file `gsoap/coordinator.h`. This header file contains datatypes and bindings described in the WSDL file. The relevant command in the *CMakeLists.txt* file is listed in Listing 4.1

Listing 4.1: Command for generating SOAP header file from WSDL

```
add_custom_command (
  OUTPUT
    ${GSOAP_DIR}/${CMAKE_PROJECT_NAME}.h
  COMMAND
    wsdl2h -o ${GSOAP_DIR}/${CMAKE_PROJECT_NAME}.h
      ${CMAKE_CURRENT_SOURCE_DIR}/../cosimulation.wsdl
)
```

In a second step, the tool `soapcpp2` is executed, which generates the implementations from the previously generated header file. The `-i` options is used to generate service proxies for C++, or in other words, code which is more suited to be used in C++. After this step, the gsoap subdirectory contains all the files needed for the SOAP server. The command as defined in the CMakeLists.txt is listed in Listing 4.2

Listing 4.2: Command for generating SOAP implementation files from the header file

```
add_custom_command (
  OUTPUT
    ${GSOAP_DIR}/soapStub.h
    ${GSOAP_DIR}/soapcoSimulationBindingService.h
    ${GSOAP_DIR}/coSimulationBinding.nsmap
  COMMAND
    soapcpp2 -i -d${GSOAP_DIR} -I${GSOAP_IMPORT_DIR}
      ${GSOAP_DIR}/${CMAKE_PROJECT_NAME}.h
  DEPENDS
    ${GSOAP_DIR}/${CMAKE_PROJECT_NAME}.h
)
```

Now the project can be compiled by adding all `.cpp` files in the `src` and `soap` directories to the compilation process, and using the subdirectories

- gsoap
- include
- include/opcuasdk
- include/opcuasdk/uaclient

- include/opcuasdk/uabase
- include/opcuasdk/uabase/arch
- include/opcuasdk/uabase/arch/gcc
- include/opcuasdk/xmlparser
- include/opcuasdk/uapki
- include/opcuasdk/uastack
- include/opcuasdk/uamodels

as source directories for header files.

### Configuration and Startup

The Coordinator supports three command-line arguments at startup:

1. `--init-file=somepath/somefile.xml` specifies a file to use for the co-simulation specification. This argument is mandatory.
2. `--xml-out=somepath/somefile.xml` specifies a file to write the co-simulation data to after the co-simulation has finished. The data will be written in XML format.
3. `--ml-out=somepath/somefile.m` specifies a file to write the co-simulation data to after the co-simulation has finished. The data will be written as matlab script.

The commands `xml-out` and `ml-out` are not mutually exclusive and can both be specified.

The Coordinator and the co-simulation it manages can be configured via an XML file given to it at startup via the `init-file` argument. This file contains initialization data for the co-simulation. An example of such a file can be seen in Appendix A 6. The root element for an init file is `cosimulation`, child elements of this are the `coupling`, `timing`, `solver` and `simulations` elements, where the first three contain global configuration data about the whole co-simulation and the `simulations` tag contains specific information about each simulation.

The coupling tag can contain the child tags

- `mode`: **Enum, Required**; One of `weak`, `dynamic-serial`, `dynamic-parallel`.

- `iterations`: **Integer, Required if precision not defined**; Number of iterations for dynamic coupling.
- `precision`: **Double, Required if iterations not defined**; Difference at which to advance to the next interval. See Section 4.3.1 for details.

The timing tag can contain the child tags

- `startTime`: **Double, Required**; Time at which to start the co-simulation.
- `endTime`: **Double, Required**; Time at which to stop the co-simulation.
- `timeStep`: **Double, Required**; Time interval between communication.
- `maxVariation`: **Double, Required**; Maximum allowed difference between coordinator-time and client-time. See Section 4.3 for details.

The solver tag can contain the child tags

- `name`: **String, Required if not overridden**; The name of the solver to be used by a simulation.
- `absoluteTolerance`: **String, Optional**; Configuration parameter for the solver.
- `relativeTolerance`: **String, Optional**; Configuration parameter for the solver.
- `minimumStepSize`: **String, Optional**; Configuration parameter for the solver.
- `maximumStepSize`: **String, Optional**; Configuration parameter for the solver.

The simulations tag contains one `simulation` tag per participating simulation, each `simulation` tag can contain the following child tags:

- `name`: **String, Required**; The name of the simulation.
- `connection-type`: **Enum, Required**; Either SOAP or OPCUA.
- `connection-information`: **String, Required if `connection-type` is OPCUA**; contains the URL of the OPC UA server, e.g. `opc.tcp://localhost:48010`.
- `ordering`: **Integer, Optional**; Used in dynamic-serial coupling, ordering is done in ascending order, meaning a simulation with odering 0 will execute before a simulation with ordering 1.
- `solver`: **String, Optional**; `solver` element which overrides global solver settings for a specific simulation.

## 4.6 The Matlab/Simulink framework

This framework is written in the Matlab scripting language and executes Simulink simulations. It communicates with the Coordinator via SOAP.

### 4.6.1 Preparation

The steps mentioned here are for understanding the toolchain used to prepare Matlab to run the framework. They are only relevant if one wishes to modify the framework or the communication pattern (WSDL/SOAP).

In order to use SOAP as a means of communicating with the Coordinator, the framework needs a SOAP client. A seemingly simple way to create such a client is to use the built-in Matlab function `matlab.wsdl.createWSDLClient(wsdlURL)`. This function takes the URL to a WSDL file as parameter and creates a WSDL/SOAP client from the definitions therein. This function, in turn, uses the Apache CXF framework [Foua]. The path to the Apache CXF framework, along with the path to a Java Development Kit has to be set in Matlab via the function `matlab.wsdl.setWSDLToolPath('JDK',jdk,'CXF', cxf)`. For the current version of the Matlab/Simulink co-simulation framework, the Apache CXF tools version 3.0.4 were used. After the `matlab.wsdl.createWSDLClient` function has executed sucessfully, Matlab should have created all the relevant files for the SOAP client in the current working directory. Unfortunately, there seem to be discrepancies between the generated Java code from the CXF framework and the generated Matlab code. To fix this, the generated Matlab file `coSimulationService.m` was modifed to work with the CXF code and saved under `CustomCoSimulationService.m`. This class has the two functions `initialize` and `registerResult` which are called by the framework to communicate with the Coordinator.

The endpoint of the WSDL service is currently configured to be the localhost on port 8080. To change this endpoint, the binding inside the WSDL file has to be changed and the `matlab.wsdl.createWSDLClient` function has to be executed to create a new client with the modified binding.

### 4.6.2 Interfacing a Model with the Framework

The Matlab/Simulink framework was designed with usability in mind. It should not be overly complicated to export datapoints to other simulations in the co-simulation setup. Likewise, it should be easy to use the datapoints of other co-simulations in the Simulink model. To achieve this, the framework handles the two things as follows:

Datapoints which are exported to the Coordinator are read from the workspace after each simulation step. This means that those datapoints have to be manually exported to the workspace from inside the model. Each datapoint that should be sent to the server

has to be explicitly exported to the workspace and marked as output in the initialization file (see Section 4.6.3 for details). This has to be done by the user for each datapoint he or she wishes to expose to other simulations.

Datapoints which are received from the Coordinator are written to the workspace before each simulation step. This means that those datapoints have to be read from the workspace from inside the Simulink model. The datapoints from other simulations are written to the Matlab workspace in the format `<simulationName>.<datapointName>`, where `<simulationName>.<datapointName>.value` contains a vector from which the actual value can be interpolated with $actual = value[1] * time + value[0]$. This interpolation has to be modelled manually inside the model itself, because only the model, at the time of simulation, has access to time information and can therefore interpolate the actual value correctly. In order to read load the datapoints from the Matlab workspace, *Constant* blocks from the Simulink components library have to be used. Others, like the *LoadFromWorkspace* block, will not work in conjunction with the framework.

### 4.6.3 Configuration and Startup

The simplest way to run the framework is to create a CoSimulation object (with the Matlab root directory at `cosim/matlab`) and give it the model file to use via the `options.modelFile` option. After creating the CoSimulation object, the co-simulation can be started by calling the function `runCosimulation()`. This will initiate the communication with the Coordinator and the execution of the co-simulation. The call will block until the co-simulation has finished or some error occurs.

Options are given to the CoSimulation object's constructor via a struct. Currently, supported fields in the options struct are:

- *initFile* Path to the XML file used at startup to determine imported/exported variables and simulation configuration options. Can be a relative or an absolute path.
- *modelFile* Path to the model file that is passed to Simulink and is used for the co-simulation. Can be a relative or an absolute path.
- *setupScript* Path to a Matlab script file that should be executed before each simulation; that is, before each invocation of the `sim(...)` function. Can be a relative or an absolute path.
- *tearDownScript* Path to a Matlab script file that should be executed after each simulation; that is, after each invocation of the `sim(...)` function. Can be a relative or an absolute path.

The framework needs an initialization file to run. When starting the framework with no initialization file specified, the framework automatically looks for a file named `init.xml`

inside the directory of the model file. If another initialization file should be used, it can be specified via the `options.initFile` option. The init-file has to be in XML format and has to have the root element `cosimulation`. Child elements of this element are `input`, which specifies which datapoints this simulation needs from other simulations, and `output`, which specifies what datapoints this simulation should send to the Coordinator. An example of such a file is shown in Appendix A 6. The `simulation` element inside the `output` element defines how the local simulation is represented to the Coordinator, i.e. the name for the simulation that is sent to the Coordinator and all the local datapoints which should be sent to the Coordinator. Datapoints within the `ouput` element contain a value that is used as an initial value in the initialization step. These values have to be modified manually if initial values in the model change. The `simulation` elements within the `input` element specify what datapoints from which simulations in the co-simulation setup are needed for the local simulation to run. Here, only the names and units of the datapoints are relevant and an initial value is not needed.

In addition to the initialization files, startup and a teardown scripts can be specified via the `options.startupScript` and `options.tearDownScript` options. The specified startup script will be executed each interval before the simulation is executed. The teardown script is executed each interval after the simulation has executed. These scripts are simply Matlab scripts that can be used for cleaning up the workspace, or other tasks that should be done regularly in-between intervals.

Listing 4.3 shows all steps necessary to run a cosimulation, provided that a Coordinator is running at the endpoint specified in the used WSDL file.

Listing 4.3: Sample code for running a co-simulation

```
matlab.wsdl.setWSDLToolPath('JDK','/opt/jdk7/', ...
  'CXF','/opt/apache-cxf-3.0.4/');
matlab.wsdl.createWSDLClient('cosim/cosimulation.wsdl');
javaaddpath('+wsdl/cosimulation.jar');
options.initFile = '/somepath/to/initfile.xml';
options.modelFile = '/somepath/to/modelfile.slx';
options.setupScript = '/somepath/to/setup.m';
options.tearDownScript = '/somepath/to/teardown.m';
cosim = CoSimulation(options);
output = cosim.runCosimulation();
```

### 4.6.4 Internal Workings

A basic class layout of the framework can be seen in Figure 4.6.

The `CoSimulation` class handles parameters and instantiates a suitable (as of now, only Simulink simulations are supported) `SimulationRunner` object. The `Simulation Runner` handles the execution of the actual simulation in the framework. In the case

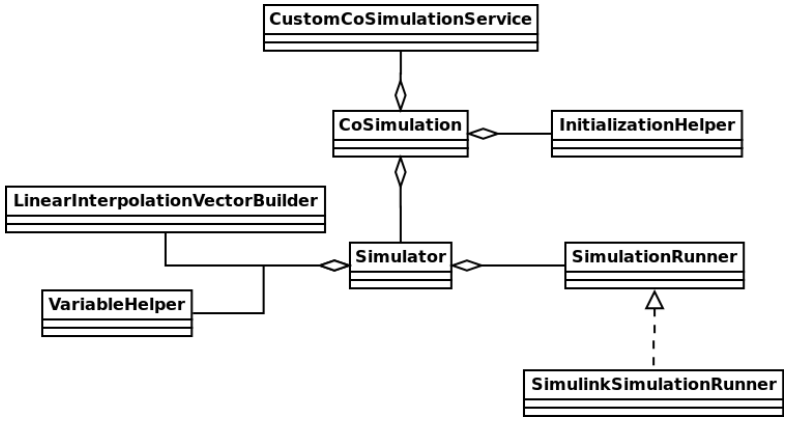


Figure 4.6: Overview of datatypes and associations in the Matlab framework

of a `SimulinkSimulationRunner`, it handles the saving of co-simulation results to the workspace, saving of the simulation state for later runs and executing Simulink to actually simulate the model. Upon calling `runSimulation()`, a `Simulator` object is instatiated with the created `SimulationRunner` and the co-simulation is started. This means that a connection to the Coordinator is opened and the initialization data is fetched from the Coordinator. This initialization data contains the macro-step size, the number of iterations, and other control data for the framework. After this initialization data is fetched, the framework begins executing the simulation and periodically sending the simulation data to the Coordinator. The response from the Coordinator can be delayed if it needs to wait for other simulations to finish. This way, the pacing of the framework can be indirectly controlled by the Coordinator.

The Matlab/Simulink framework works by saving the complete end-state after each simulated interval and then starting the next interval with the previous state object. Simulations are not paused, but run only for one interval at a time, a simulation is completely restarted for each interval. This approach is more resource and time intensive, but ensures that results from other co-simulations are loaded from the workspace for each interval and also enables dynamic coupling by being able to restart an interval with other input values.

To summarize, the following enumeration lists the steps taken by the framework in each iteration of the co-simulation.

1. Calculate the interpolation vectors from the step-start-results and the current results.
2. Run a user-defined setup script (in this case just setting some parameters in the workspace).
3. Execute the `sim(...)` command.

4. Save the simulation result to the workspace.

5. Run a user-defined tear-down script (in this case deleting the parameters from the workspace).

6. Get variable values from the final state.

### 4.6.5 Problems and Difficulties

A big problem, encountered early on, was that the WSDL client, which Matlab automatically creates with a call to `matlab.wsdl.createWSDLClient(wsdlURL)`, seems to require a specific version of the Apache CXF framework [Foua] to work. The used version 3.0.4 does not produce errors when creating the client, but when executing SOAP calls, some internal calls to the Java code in the background fail because of mismatched datatypes. Because of this, the generated matlab WSDL client had to be modified manually in order to get it working properly.

Another problem, which is still not resolved, is that the automatically created client is not able to translate SOAP faults correctly into Matlab. This means that each time a fault is sent from the Coordinator to the Matlab framework, it just fails with a generic exception inside the Java code instead of a managable exception in the Matlab code.

In order to get dynamic coupling working with Simulink, the state of the simulation has to be saved across iterations. This proved to be a very time-consuming endeavor. The state of a simulation in Simulink can be saved in two different ways, partial or complete. A partial simulation state can be saved in Simulink by passing the *'SaveFinalState', 'on'* to the `sim` command. When passing these two parameters, Simulink will save a subset of the final simulation states to the workspace. These states, however, do not contain the internal information of integrators and other blocks that rely on internal states, like Stateflow charts. In order to save all states, the parameters *'SaveCompleteFinalState', 'on'* have to be passed to the `sim` command as well. While only saving the final state allows for small modifications of the model between stops and restarts, saving the complete final state forbids any modifications of the model in order for the state to be restored. This behaviour proved to be problematic in conjunction with some building blocks, most of all the *LoadFromWorkspace* block. This block works when only saving the partial final state, but generates errors when using the complete final state and the value of the workspace variable changes between simulation runs. After some trial and error, it was discovered that *Constant* blocks from the Simulink component library also have the ability to load variables from the workspace and do not generate errors if the value changes between runs. The difference being that Constant blocks only load the value of a workspace variable at the very start of the simulation run. This, however does not impact the functionality of the co-simulation framework, and so, all *LoadFromWorkspace* block could easily be replaced by *Constant* blocks.

## 4.7 The OPC UA Server & OpenModelica Framework

The OpenModelica framework, along with the OPC UA server is written in C++. It communicates with the Coordinator via OPC UA binary encoding and creates simulation binaries via the OpenModelica compiler. Figure 4.7 shows the most important classes of the framework and their associations.

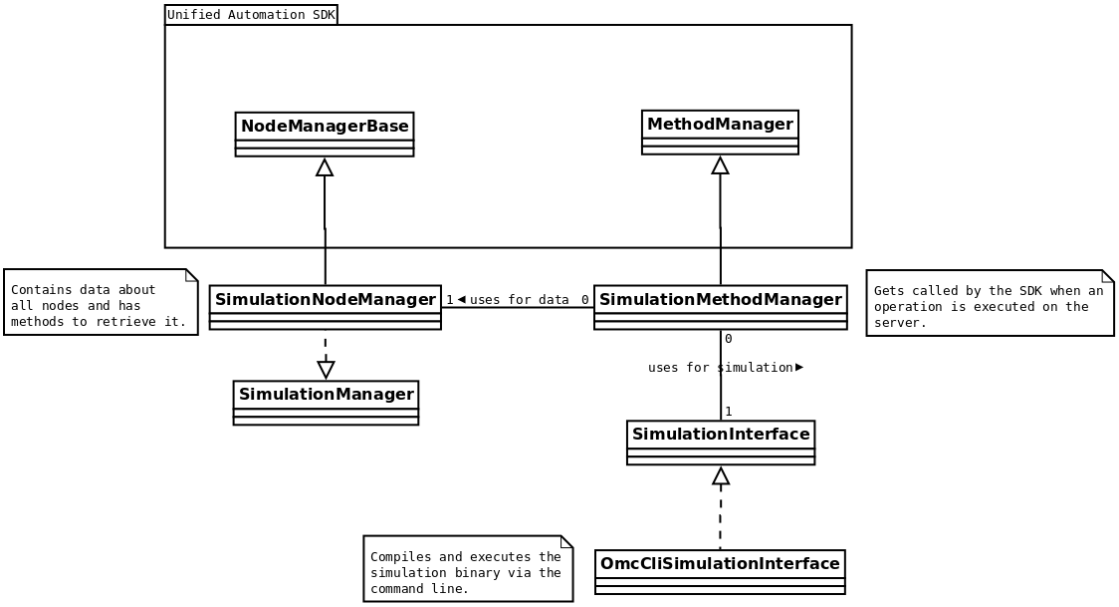


Figure 4.7: Most relevant classes of the OpenModelica framework

### 4.7.1 Preparation & Build Process

Like with the Coordinator, the C++11 standard is used for compiling/writing the OpenModelica framework. Again, `cmake` is used as a build automation tool. In order to parse XML initialization files, the `TinyXML2` library is used again. As in the Coordinator, the Unified Automation [Aut] SDK is used in conjunction with the free evaluation license to create an OCP UA server and communicate with the Coordinator. Because of this, the addiational libraries *xmlparser*, *xml2*, *ssl* and *crypto* are needed again as well. In addition to those libraries, the *boost* libraries *system*, *program_options* and *filesystem* are used and needed for compilation. Attention: the OPC UA server caused a segmentation fault at startup when compiled with gcc >= 5.0. The problem seems to lie in the Unified Automation libraries and can thus not be circumvented, gcc 4.8 is recommended for compilation of the project.

#### The Build Process

The steps described here are for understanding only, they are automatically executed when `make` is run in the *openmodelica* directory. All the commands and steps are integrated and executed via the *CMakeLists.txt* file. For convenience, a custom *makefile* was created to automatically build the project out-of-source in the *build* directory.

To compile the OPC UA server, a C++11 capable compiler has to be used or the compilation will fail. If a GNU [Foub] C++ compiler is found on the compiling system, the `-std=c++11` flag is set automatically. If the compilation process finishes correctly, the executable for the OpenModelica framework should be located at *openmodelica/build/server/opcuaserver*.

### 4.7.2 Interfacing a Model with the Framework

The only difference between a normal OpenModelica model and a model specifically created for a co-simulation is the handling of the interpolation vectors which are received from the other simulations in the co-simulation setup. Such vectors arrive in the form $value_t = value[1] * t + value[0]$ where $value_t$ is the value of a variable at time $t$ and $value[]$ is the vector of constants. To differentiate between different co-simulations, each variable from a co-simulation is prefixed with the co-simulation's name and has to be used that way. For example, in the model created for this thesis, the OpenModelica simulation requests the value $TZ1\_Pel\_O$, which stands for *Thermal Zone 1 - electrical Power - Oven*, from the simulation *thermisch*. Thus, to use the variable in a model, it has to be declared as follows:

Listing 4.4: Co-simulation-variable declaration in an OpenModelica model

```
model Machines:
  ...
  Real[2] thermisch__TZ1_Pel_O;
  ...
```

As can be seen, the variable is prefixed with the name of the simulation *thermisch* and two underscores. After the prefix comes the name of the variable as it is presented by the *thermisch* simulation. This way the framework "injects" the value of this variable into the OpenModelica simulation at the start of each interval.

In order to get the actual value of a variable and to balance out equations for OpenModelica, three equations are necessary:

Listing 4.5: Necessary equations for co-simulation-variables in an OpenModelica model

```
  equation:
    thermisch__TZ1_Pel_O[1] = pre(thermisch__TZ1_Pel_O[1]);
    thermisch__TZ1_Pel_O[2] = pre(thermisch__TZ1_Pel_O[2]);
```

```
oven.P_el.p = thermisch__TZ1_Pel_O[2] * time +
  thermisch__TZ1_Pel_O[1];
...
```

The first two of those equations balance out the equations in the simulation and ensure that the values of the given constant vector stays the same. The second equation calculates the actual value of the variable *thermisch__TZ1_Pel_O* at time *time* and assigns it to the correct variable in the model.

### 4.7.3 Configuration and Startup

Like the Matlab framework, the OpenModelica framework expects an XML configuration file at startup, given to it via the *config* argument. This file contains information about imported and exported datapoints and initial values of exported datapoints. See Section 4.6.3 for more information about the XML config file. In addition to this co-simulation configuration file, the OPC UA framework expects a configuration file for the OPC UA server. This file has to be present in the root directory of the framework binary. An example file can be found at *opcua/server/serverConfig.xml*, this file is copied into the *opcua/build/server/* directory at build time so that the binary can be executed.

### 4.7.4 Internal Workings

Internally, the framework consists of an OPC UA server in the simulation part, and an OPC UA client in the Coordinator. When the simulation part of the framework starts, it will first read the initialization file given to it as an argument. After it has processed which variables to export and import, it will generate those variables in the OPC UA path tree. Figure 4.8 shows how the nodes in the OPC UA server are arranged.

Directly below the root element in the tree are the three main nodes which are relevant for co-simulation: *configuration*, *cosimulations* and *simulation*. The *configuration* node is of type `SimulationConfigType`. It contains child objects of types `SolverConfigurationType` and `TimingConfigurationType` which contain datapoints for the solver and timing settings respectively. Figure 4.9 shows an overwiew of the type hierarchy and composition. The configuration node is written to by the Coordinator in the initialization phase.

The *cosimulations* node contains two children, the *step_start* and the *step_end* nodes. Each of these two nodes contains the state of all co-simulations. Each simulation node is named after the simulation as configured in the initialization file and is of type `SimulationType`. A `SimulationType` object contains `DatapointType` child objects named after the respective datapoints of the simulation. The *step_start* node contains the state at the start of the interval, and the *step_end* node contains the most recent state. See Section 4.3.1 (step-start-results) for details. These nodes are written by the Coordinator before each simulation step.

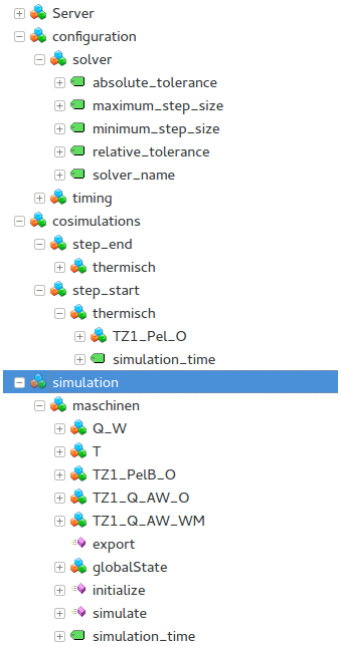


Figure 4.8: Node Tree layout of the OPC UA server

The last node, *simulation*, contains data about the current simulation which the OPC UA server is managing and executing. This node not only contains the exported datapoints, but the three operations: `initialize()`, `simulate(startTime, endTime)` and `export()`.

When the Coordinator is in the initialization phase, all existing `OpcUaSimulationConnectors` will write configuration settings and call the `initialize()` operation of their respective OPC UA servers. When this operation is called on the server, it will compile the model by calling the `omc` tool (the OpenModelica compiler). After the initialization step, the Coordinator reads the initial datapoint values from the server.

The `simulate(startTime, endTime)` operation is called for each step in the co-simulation. Before each call to this operation, the client in the Coordinator sets the respective datapoint values of the simulations in the OPC UA path tree of the server by issuing a write command. It then executes the operation and the server simulates the given interval with the co-simulation values set. After the simulation step, the Coordinator reads the simulation's datapoint values from the path tree of the server by issuing a read command. The read and write commands from the Coordinator to the simulation are executed in bulk in order to save bandwidth and increase speed.

The `export()` operation is used at the end of a simulation in order to write the history of the simulation to a matlab file named *results.m* in the execution directory of the OPC UA server.

To compile the model in the initialization step, the framework uses a template file

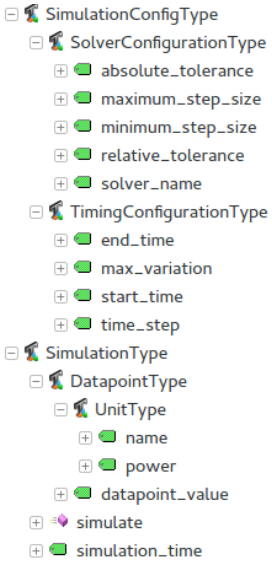


Figure 4.9: OPC UA type definition

(*compilation_template.mos* located in the *opcua/server/* directory). It replaces the values in the template file and copies this file as *compile.mos* into the directory of the model to simulate. After the template is copied, the command `cd <model-directory> && omc compile.mos` is executed to compile the model. This will generate the simulation binary inside the *build* subdirectory. As can be seen in the template file, the output of the simulation binary is set to CSV in order to be able to read the results more easily. To be able to start the OpenModelica simulation with different values, the framework utilizes the *overrideFile* argument of the simulation binary. After each step, it reads the CSV values generated by the simulation binary in the result file *<modelname>_res.csv*. Before each simulation step, it generates an override file based on the last results in the result file and the co-simulation values set by the Coordinator. The simulation binary is called as follows: `cd <compilation-directory> && ./<simulation-binary> -overrideFile=<override-file>`

The sequence of events in each iteration of the co-simulation is as follows:

1. Calculate the interpolation vectors from the step-start-results and the current results.

2. Write the override-file with parameters and variable values to disk.

3. Run the simulation binary with the override-file as parameter.

4. Parse the result file and update variable values in-memory.

### 4.7.5 Problems and Difficulties

Although the integration of the OpenModelica framework into the co-simulation setup was much simpler than the Simulink framework, it did not come without problems.

The first problem was the selection of an OPC UA SDK. Since virtually all existing SDKs were only available commercially, the decision fell to the Unified Automation SDK because it offers one hour of functionality per run. A major problem was encountered later on when using gcc version >= 5.0 for compiling the project. It seems that these versions of the compiler cause a problem with the Unified Automation SDK libraries where the server cannot be started and a segmentation fault occurs when trying to do so. It is therefore recommended to use gcc 4.8 to compile the framework.

Like with the Simulink framework, imported variables actually need to consist of two parts, an offset and a gradient. The gradient is always zero in non-dynamically coupled simulations because no previous results are available. In dynamic coupling, however, these two parts are necessary in order to linearly interpolate a value of an imported variable over time. Offset and gradient are based on the results of the last interval and the results of the current interval in a previous iteration.

Initializing the OpenModelica simulation with new values proved more difficult than anticipated. Early attempts tried to modify the *<modelname>_init.xml* file created by the compilation process, without success. After some trial and error, the `-overrideFile` parameter was discovered. This parameter can be used when running the simulation binary, it specifies a comma-separated file with new values for simulation parameters, as well as all variables occurring in the model. With this parameter, the framework could be implemented as intended. The problems with this approach are, that each iteration of a simulation step generates a new file on the hard drive and that writing and subsequent reading of the file by the simulation binary generates overhead that would not be necessary if the passing of parameters could be done in-memory.

CHAPTER 5

# Testing & Evaluation

## 5.1 Implementation vs. Requirements

### 5.1.1 Functional Requirements

#### 5.1.1.1 Requirement 1: Support for Weak Coupling

Support for weak coupling has been implemented. It can be chosen by setting the coupling mode in the Coordinator initialization file to `weak`.

#### 5.1.1.2 Requirement 2: Support for Dynamic Parallel Coupling

Support for dynamic parallel coupling has been implemented. It can be chosen by setting the coupling mode in the Coordinator initialization file to `dynamic-parallel`. The number of iterations can be chosen by setting the iterations in the Coordinator initialization file to the desired amount. The initialization files of the simulations do not need to be modified.

#### 5.1.1.3 Requirement 3: Support for Dynamic Serial Coupling

Support for dynamic serial coupling has been implemented. It can be chosen by setting the coupling mode in the Coordinator initialization file to `dynamic-serial`. The number of iterations can be chosen by setting the iterations in the Coordinator initialization file to the desired amount. The order in which the simulations are executed can be set by giving the simulations inside the Coordinator initialization file an `ordering` value. The initialization files of the simulations do not need to be modified.

#### 5.1.1.4 Requirement 4: Support for Units as Metadata

Units are sent from the simulation frameworks to the Coordinator. The Coordinator aborts the co-simulation if units for the same datapoint are different and it cannot automatically convert between them.

#### 5.1.1.5 Requirement 5: A Central Server for Data Exchange

Has been implemented as the *Coordinator*.

a) *Support multiple break conditions for dynamic coupling.* The Coordinator supports a maximum number of iterations and a minimum error that has to be reached in order to advance the simulation. Those limits can be set via the `iterations` and `precision` elements inside the Coordinator initialization file.

b) *Support an ordering of simulations for dynamic serial coupling.* Has been implemented via an XML element `ordering` inside the `simulation` elements of the Coordinator initialization file.

c) *Set initialization data.* The initialization file for the Coordinator allows for setting the start and end time of a co-simulation and an interval between data exchanges via the `timing` element and its child elements in the initialization file. Furthermore, a global solver to be used and specific solvers for each participating simulation can be set via the global `solver` element and the individual `solver` elements of the simulations.

d) *Convert between units.* The Coordinator automatically converts between different powers of the same unit. It is extendable by implementing a new class derived from the common base class, see Section 5.1.2.4 for details.

e) *Support SOAP data transfer.* The Coordinator contains a SOAP server in order to receive and transmit SOAP data.

f) *Support OPC UA data transfer.* The Coordinator contains an OPC UA client in order to receive and transmit OPC UA data.

#### 5.1.1.6 Requirement 6: A co-simulation framework for Matlab/Simulink

Has been implemented in Matlab as the *Matlab framework*.

a) *SOAP client for communicating with the server.* The Matlab framework contains a SOAP client in order to receive and transmit data from an to the Coordinator via SOAP.

b) *Ability to execute Simulink simulations.* The Matlab framework has the ability to execute Simulink simulations. It can insert data into them and extract data from them in order to exchange data with the Coordinator.

#### 5.1.1.7 Requirement 7: A co-simulation framework for OpenModelica

Has been implemented in C++ as the *OpenModelica framework*.

a) *Ability to communicate via OPC UA.* The OpenModelica framework contains an OPC UA server in order to receive and transmit data from an to the Coordinator via OPC UA.

b) *Ability to execute OpenModelica simulations.* The OpenModelica framework can execute OpenModelica simulations and insert/extract data into/from the simulations to exchange it with the Coordinator.

### 5.1.2 Non-Functional Requirements

#### 5.1.2.1 Speed

The speed of the co-simulation can be measured in two distinct ways. First, the time it takes a simulation tool to simulate a model from a start time $t_s$ to and end time $t_e$, and second, the time it takes for data to be exchanged (transmitted and received) with the Coordinator. To be able to measure these two metrics, a timing functionality was implemented in both the Coordinator and in each framework (Matlab/SOAP and OpenModelica/OPC UA). The Coordinator times how long it takes for a simulation framework to execute each timestep and iteration. The result of this measurement is a list of millisecond-durations for each connected simulation. Each of these lists contain the durations between registering new result, i.e. the duration of one simulation of the timestep and the duration of the data transfer to/from the coordinator.

The results of this measurement can be seen in Figure 5.1. For this measurement, the *thermisch-maschinen* co-simulation was simulated with settings for Scenario 1 from time 0 to 30000$s$. The *thermisch* simulation was simulated by the Matlab framework and the data transferred via SOAP to the Coordinator. The *maschinen* simulation was simulated by the OpenModelica framework and the data transferred via OPC UA. As can be seen right away, there is quite a difference in the duration of the simulation runs between the two used technologies. To identify the causes for such a huge time difference, however, the simulation times and the data transport times have to be separated and viewed independently of each other.

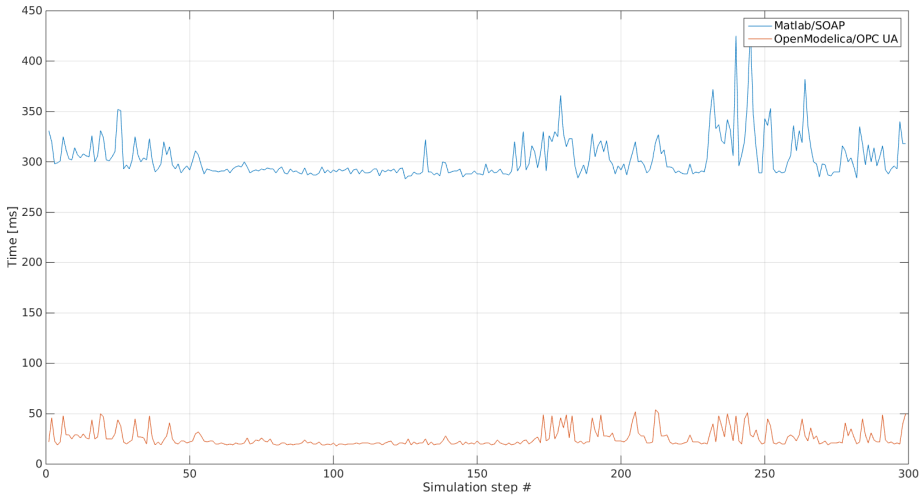


Figure 5.1: Duration of simulation runs measured by the Coordinator

### Co-Simulation Speed

In order to measure the time it takes for the Coordinator to process the simulation data, a separate timing mechanism was implemented. The Coordinator times each major data processing step in each request. The complete time taken per request is recorded and output as Matlab data. In the tested scenario, two simulations exist, where one (Matlab) sends eight and requests four datapoints and the other simulation (OpenModelica) sends six and requests one datapoint. Figure 5.2 shows that data processing takes about the same time for both simulations, although the mean value of the times of the Matlab simulation is quite a bit higher than the mean value of the times of processing durations of the OpenModelica simulation. While the Matlab simulation with eight sent and four requested datapoints takes an average of 1.65 ms to process one request, a request from the OpenModelica simulation only takes an average of 1.03 ms to process on the Coordinator. This behavior can probably be explained by the structure and the complexity of the code. In a parallel co-simulation, the Coordinator builds the step-start-results once every timestep. This process has a worst-case complexity of $O(N * M * K * L)$ where $N$ is the number of simulations that participate in the co-simulation setup and $M$ is the number of simulations which provide at least one datapoint to other simulations. $M$ is equal to $N$ if every simulation requires datapoints from all other simulations and every simulation provides at least one datapoint, so the complexity can be simplified to $O(N^2 * K * L)$. $K$ is the number of datapoints that a simulation requires from another simulation and $L$ is the number of datapoints that the other simulation provides. In addition to the step-start-results, the Coordinator builds the current results for an iteration once every iteration. This process has the same worst-case complexity of $O(N^2 * K * L)$. The code in the Coordinator is written so that the last thread to receive data from a client is the first

one to execute those data processing steps. Assuming that the Matlab simulation takes longer to simulate and send data than the OpenModelica one (see below), the Matlab thread in the Coordinator is always the first thread to execute the data processing. All following threads only have a complexity of $O(logN)$ for retrieving the needed data from a map. In contrast to the parallel co-simulation, a serial one would have a complexity of $O(N^2 * K * L)$ once every timestep and a complexity of $O(N * K * L)$ for every thread in every iteration and timestep.

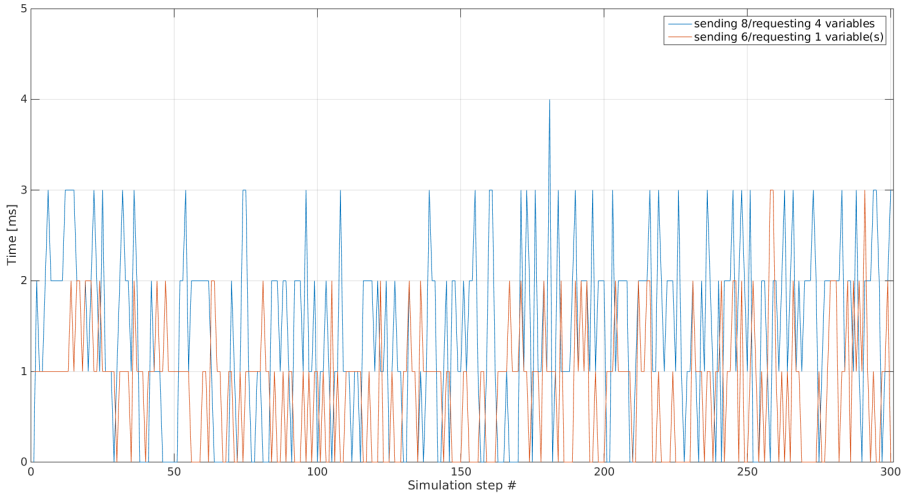


Figure 5.2: Duration of data processing in the Coordinator

To measure simulation time, each of the frameworks take measurements of the time it takes to setup/tear down the simulation process and the actual time to simulate the model itself. The results of these measurements is a list per framework which contains the durations of each simulation run. A comparison between Matlab and OpenModelica can be seen in Figure 5.3. It is apparent that Matlab/Simulink takes about 7 times longer to simulate the *thermisch* part of the co-simulation as OpenModelica takes to simulate the *maschinen* part. This outcome might be based on several factors. First, it might be that the models simply take a different time to simulate due to their different complexity. Second, the initialization process and the code execution of the Matlab framework take longer than the OpenModelica C-binary to execute. See Sections 4.6.4 and 4.7.4 for a coarse-grained list of events which happen in each iteration cycle that could influence the simulation speed.

To further examine where the huge time difference comes from, another measurement was taken solely for execution of the matlab `sim(...)` function, the results are also displayed in Figure 5.3. As can be seen, the `sim(...)` function alone takes up a huge chunk of the time needed to simulate a step. The rest of the framework execution also takes about $50ms$, but in contrast to the simulation itself, this duration is nearly constant.

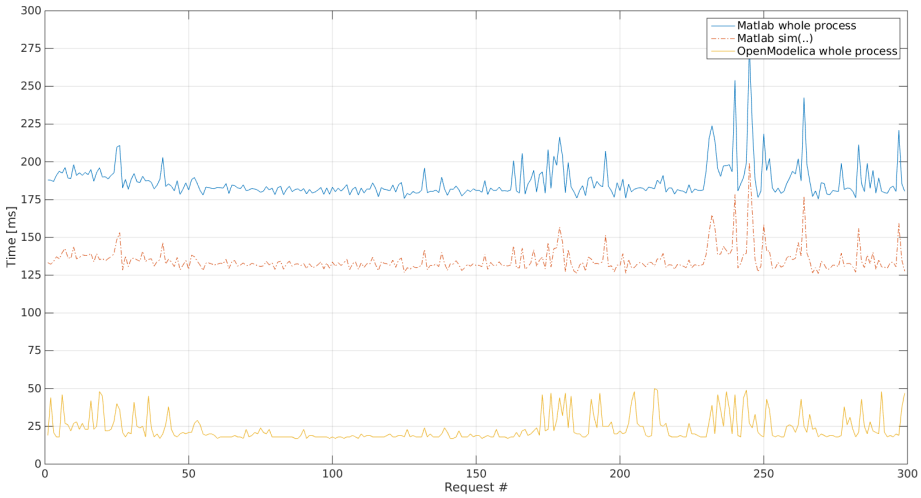


Figure 5.3: Duration of the Matlab sim(...) function vs the whole process and OpenModelica

### Data Transfer Speed

To calcualte the time needed to transfer data to and from the Coordinator, the timings from the Coordinator and from the frameworks were used. The transfer time was calculated by subtracting the timings of the frameworks from the timings of the Coordinator. As an example, let $t_{mc}$ be the time the Coordiator measured for a Matlab/SOAP simulation and $t_m$ be the time that the Matlab/SOAP framework itself measured for the execution of a simulation step. The time it takes for the SOAP connection to send and receive data is then $t_{tr_s} = t_{mc} - t_m$. To clarify further, the Coordinator starts a timing $t_{mc}$ right before it sends SOAP data back to a client and stops the timing right after it received the next batch of data from the client. This time therefore includes the time it takes for the data to be sent to the client/framework, the time $t_m$ it takes for the framework to simulate one step, and the time it takes for the framework data to be sent to the Coordinator. If the time $t_m$ is then subtracted from the time the Coordinator measured ($t_{mc}$), the data transfer time can be calcuated. Figure 5.4 shows how SOAP data transfer performs in contrast to OPC UA binary transfer. Not only is SOAP far slower (about 20 times), it also contains significant spikes which reduce transfer speed even further.

In order to analyze further, Wireshark [tea] was used to capture the data sent to/from the Coordinator.

In the case of SOAP, one request to the Coordinator, sending eight datapoints was about 1560 bytes in size. This number also contains the fixed part of the request which does not scale with the amount of datapoints sent. Using this number and ignoring the fixed part

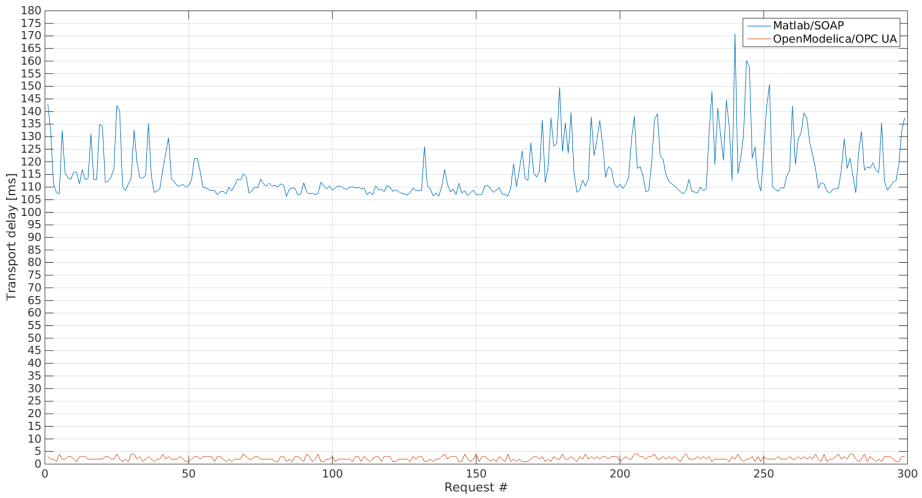


Figure 5.4: Duration of data transfer, SOAP vs OPC UA

of the request, transferring one datapoint to the Coordinator via SOAP takes roughly 200 bytes of data. A response from the Coordinator, effectively transferring four datapoints, is about 2660 bytes in size. The response, however, contains each datapoint twice because of the step-start-results. Due to this, every datapoint sent from the Coordinator should take about twice the amount of data than a datapoint in the request. Calculating the bytes per datapoint for the response (2260/4) yields 565, which is nearly triple the size of a datapoint in the request. Upon closer inspection it was found that gSOAP, the SOAP tool used in the Coordinator, writes namespaces more detailed than the Apache CXF framework. As a result, one request/response cycle with eight datapoints sent to the Coordinator and four datapoints received from the Coordinator needs about 4220 bytes in transfer volume. The amount of data needed for $x$ datapoints sent and $y$ datapoints received can be calculated as follows: $bytes\_needed = x * 200 + y * 565$.

The case is a little bit more complex when looking at OPC UA, as there are more short requests and other small transfers in between the actual data transfer. As the communication is implemented right now, the Coordinator will:

1. Write the step-start-results to the OPC UA server.
2. Call `simulate()` on the server.
3. Read all needed string values from the server in bulk.
4. Read all needed numerical values from the server in bulk.
5. Write the co-simulation results to the server.

This leads to a number of request/response cycles which can be further broken down as follows:

1. 1 datapoint: 217 bytes, ack from server: 130 bytes, whole cycle: 347 bytes, worst case: 347 bytes/datapoint.
2. method call: 226 bytes, ack from server: 154 bytes, whole cycle: 380 bytes, worst case: 380 bytes/datapoint.
3. Request for reading 6 variable unit names: 502 bytes, response: 267 bytes, whole cycle: 769 bytes, about 128 bytes/datapoint.
4. Request for reading 6 variable values, 6 unit values and the simulation time: 959 bytes, response: 440 bytes, whole cycle: 1399 bytes, about 233 bytes/datapoint.
5. 1 datapoint: 215 bytes, ack from server: 130 bytes, whole cycle: 345 bytes, worst case: 345 bytes/datapoint.

One such request needs about 3240 bytes in total. When sending $x$ datapoints to the OPC UA server and querying $y$ datapoints from it, the amount of bytes transferred can be roughly calculated as follows: $x * (347 + 345) + 380 + y * (128 + 233)$.

The comparatively huge amount of data needed for OPC UA stems mainly from using strings as node identifiers and that each node identifier is a concatenation of parent node name and own name. For example, the identifier `simulation.maschinen.TZ1_Q_AW_WM.datapoint_value` selects the value node of the datapoint node `TZ1_Q_AW_WM` belonging to the `maschinen` simulation. A request contains one such string for each value that is requested from the server. The fact that OPC UA is nontheless considerably faster than SOAP may result from the parsing and conversion overhead of XML and protocol overhead of HTTP.

#### 5.1.2.2 Usability

Usage of the co-simulation setup is intended to be as straight-forward as possible. The Coordinator and the Matlab/OpenModelica frameworks are configured via XML files which are passed to them at startup and define which datapoints from other simulations are required and which datapoints from the local simulation are exported. Simulation configuration parameters can also be set locally for every simulation, but are potentially overridden by the Coordinator configuration.

The used models themselves need to be slightly adapted in order to work with the co-simulation setup. Suppose the setup consists of a Simulink simulation $A$ and an OpenModelica simulation $B$. Simulation $A$ uses the datapoint $B.x$ and simulation $B$ uses the datapoint $A.y$. For this to work, Simulation $A$ has to export datapoint $y$ and

simulation $B$ has to export datapoint $x$ and both have to import the datapoint of the other simulation.

In Simulink, importing is done via a constant block and requires an additional interpolation step before the value can be used in the simulation. Exporting a value to the Coordinator requires the model to explicitly write the value to the workspace after the simulation, e.g. via a *ToWorkspace* block. In OpenModelica, importing is done similarly via the declaration of an array inside the model and also requires an interpolation step before the value can be used. Exporting a value in an OpenModelica simulation does not require any additional steps. This means that modifying an OpenModelica model for use in the co-simulation setup is slightly less intrusive and time-intensive than modifying a Simulink model.

Starting the individual components differs slightly in complexity.

Starting the Coordinator is pretty straight-forward, it only requires an initialization file. Optionally, XML and Matlab output files can be specified if needed. The initialization file structure for the Coordinator is slightly different for SOAP and OPC UA connections. The SOAP endpoint is defined in the WSDL and thus can only be altered by changing the WSDL and recompiling the Coordinator. The OPC UA server address is defined in the initialization file and can be changed easily without recompilation. So, while it is easier to change the OPC UA server address, it requires slightly more configuration inside the initialization file. On the other hand, SOAP requires slightly less configuration, but changing the server endpoint requires a recompilation of the Coordinator.

The same principle goes for the frameworks. The Matlab/SOAP framework does not require a server address to be specified, as that is taken from the WSDL file and hardcoded into the code generated by the Apache CXF framework. If the server address needs to be changed, the whole WSDL client code has to be generated again. The OpenModelica/OPC UA framework's server port is specified inside the *ServerConfig.xml* file and can easily be changed without the need for a recompilation. The downside of this is, however, that the framework cannot be started without both, an initialization file for the co-simulation configuration, and a configuration file for the OPC UA server, located inside the directory of the framework binary.

To summarize, we look at specific scenarios and the steps that need to be taken to make them work.

#### Import a value in a Simulink simulation

In order to use a value from another simulation inside a Simulink simulation, the following steps have to be taken:

- Declare the need for the new datapoint in the `input` tag inside the *init.xml* file.

- In the model, create a constant block with its value being the name of the new datapoint, datapoints have names of the form `simulationName.datapointName`. The constant block will then provide a 2-dimensional vector to the simulation, with the first value being the value of the datapoint at the start time and the second value being the value at the end of the current timestep (for dynamic coupling).
- Linearly interpolate the value for the new datapoint, e.g. using a function block and calculating $u(2) * t + u(1)$.

#### Export a value from a Simulink simulation

To supply a value from a Simulink simulation to the other simulations in the setup, the following steps have to be taken:

- Declare that the simulation is exporting the variable in the `output` tag inside the *init.xml* file.
- In the model, create a `toWorkspace` block and give the workspace variable the same name as in the *init.xml* file.

#### Import a value in an OpenModelica simulation

To import a shared variable inside an OpenModelica simulation, simply:

- Declare the need for the new datapoint in the `input` tag inside the *init.xml* file.
- Create a 2-dimensional `Real` vector which has the name of the new datapoint, datapoints have names of the form `simulationName__datapointName`. This vector will be initialized with the value of the other simulation when the co-simulation starts.
- Create a new `Real` value which holds the actual interpolated value of the datapoint.
- Linearly interpolate the value for the new datapoint, e.g. create a new equation which interpolates the values of the vector and assigns the result to the actual variable.

#### Export a value from an OpenModelica simulation

The only thing that needs to be done in this case is to declare the export of the datapoint inside the *init.xml* file.

**Add a new simulation to the co-simulation setup**

To add a whole new simulation to a co-simulation,

- Create an initialization file, set properties and declare imported and exported datapoints.
- Modify the model to handle the import and export of datapoints.
- Modify other models to import datapoints of the new simulation.
- Add an entry for the new simulation inside the *init.xml* file of the Coordinator.

#### 5.1.2.3 Scalability

Due to the fact that the communication between the components of this project is based on standard network protocols, multiple machines can be used in parallel to run one co-simulation. The bottleneck in the setup will eventually become the Coordinator, as it is the central component where all the data and connections meet. Specifically, either bandwidth or processing power/memory will eventually slow down the co-simulation. For such a thing to happen though, a very high number of individual simulations will need to be participating in one co-simulation.

Due to the fact that the worst-case complexity of the Coordinator for the first thread in each timestep is $O(N^2 * L * K)$ (see Section 5.1.2.1), this processing step might become a processing-power bottleneck if a fair amount of simulations and exchanged datapoints are involved in the co-simulation setup. Of course, a high number of exchanged datapoints also means that more bandwidth is used and more time is needed for transferring the data to and from the Coordinator. Basically, the scalability, or better, what will become a bottleneck first, highly depends on the hardware setup.

#### 5.1.2.4 Extendability

The Coordinator can easily be extended with another protocol for data transfer, all that has to be done is to implement another `SimulationConnector` (*include/simulation/simulation_connector.h*) class for the new protocol. The new class needs to implement the `Start(Simulator*)` and the `Shutdown()` functions, where the `Start(Simulator*)` function is called by the `Simulator` when the co-simulation begins. The `Simulator*` argument is used as a callback, two functions have to be invoked by the new component in order for it to work properly. First, the `InitializeSimulationOrDie(InitRequest)`. This function of the `Simulator` has to be called once from within the `Start()` function of the new `SimulationConnector` implementation, after the connected simulation has been initialized and the data for an `InitRequest` is present. The function call will block until the Coordinator has received

data from each simulation and is ready to start the co-simulation. The function returns an `InitResponse` which contains configuration data for the connected simulation based on the initialization file. After the connected simulation is initialized, the second function of the `Simulator`, `RegisterResultOrDie(RegisterResultRequest)` has to be invoked repeatedly until the co-simulation has finished. It returns a RegisterResultResponse object which contains data about the other simulations in the setup. This function might block for some time, depending on what coupling mode is used in the co-simulation. If dynamic-serial coupling is used, it might block for the time it takes all other simulations in the setup to execute one simulation step.

The Matlab framework can also quite easily be extended for other types of simulations besides Simulink. To extend the Matlab framework, another `SimulationRunner` has to be implemented for the new simulation type. The type of `SimulatonRunner` is set in the constructor of the `CoSimulation` class and is currently fixed. If new `SimulationRunners` are implemented, some kind of selection for the `Simulation Runner` (maybe based on a setting in the initialization file) would be necessary.

The OPC UA-based framework can also easily be extended to execute simulations other than OpenModelica. All that is needed is to implement another `SimulationInterface` (*simulation/simulation_interface.h*). Currently, the `SimulationInterface` is set in the *server_main.cpp* file. Here, it would also be a good idea to select the `Simulation Interface` to be used based on a setting in the initialization file.

Adding new datatypes for data management and transfer has to be done in-source. To add a new datatype to the Matlab framework, the WSDL file has to be modified to contain the new datatype. After the WSDL has been modified, the Matlab command for creating the SOAP client (`matlab.wsdl.createWSDLClient(wsdlURL)`) has to be run again in order to create the necessary Java code. Some modifications inside the *CustomCoSimulationService.m* file and the rest of the framework may be necessary to incorporate the new datatype into the framework. To use the new datatype in the Coordinator, the best option is to modify *soap/soap_util.cpp*. The functions in this file are used to convert *gSOAP* datatypes into the ones used in the Coordinator code. After the necessary changes are made, a recompilation is sufficient to rebuild the *gSOAP* code based on the new WSDL file. Adding a new datatype to the OpenModelica framework simply entails transporting the data of the datatype to and from the Coordinator as simple types like integer or double and recreating the desired type on the other side.

The unit converter inside the Coordinator currently only converts between exponents of the same base unit. In order to implement a more sophisticated converter, a new `UnitConverter` (*conversion/unit_converter.h*) has to be implemented and set inside the constructor of the `CoSimulation` class. As multiple converters become available, it might be convenient to add a mechanism to select the converter based on a setting in the initialization file.

## 5.2 Scenario Results

The co-simulation environment was tested with a proof-of-concept model originating from the *Balanced Manufacturing* (BaMa) project [Rai+16]. The original model modells a bakery and has real-world applications. The model used here is has been simplified so as to allow for easier splitting and handling. The modelled factory consists of four individual thermal zones. Thermal zones 3 and 4 exist only to transfer thermal energy between the rooms inside the building. Thermal zone 1 contains all the machines used for production, whereas thermal zone 2 contains the heating, cooling and electrical components. These components supply the machines with thermal and electrical power and are responsible for a stable temperature of the thermal zones themselves. The production machines process discrete entities. The heating of the machines influences the ambient temperature surrounding them, which in turn, influences the temperature of the other thermal zones. The **Oven** contains a two-point controller with hysteresis which regulates its temperature and a conveyor belt which transports the entities from the input to the output. The conveyor belt can transport multiple entities at the same time, therefore allowing multiple entities to be inside the **Oven** simultaneously. The model of the oven component as well as thermal zones and energy supply are not described in detail here, more information can be found in [Rai+16], [Püh16] and [Smo+16].

The derived proof-of-concept model is split into two parts, a *machinen* and a *thermisch* model. As the name suggests, the *machinen* model handles all the functionality of the machines, this case only handles the material flow including the **Oven**. The *thermisch* model handles heat, cold and electrical energy transfer in the building. An overview of the reduced model can be seen in Figure 5.5. The **Oven** receives entities from an **EntitySource**, processes them and outputs them into an **EntitySink**. The amount of energy needed and the amount of waste-heat produced by the **Oven** are sent to the *thermisch* model, the thermisch model processes these values and sends the amount of energy that the **Oven** requested back to the *maschinen* model. A simplified illustration can be seen in Figure 5.6. When the **Oven** needs energy from the *thermisch* model, it can request that energy only at a macro-step, when communicating with the *Coordinator*. This introduces a delay that can be used to test and compare different coupling strategies and macro-step sizes in terms of accuracy. As part of the validation, the results of the co-simulations will be compared to the results of the BaMa project in the following scenario-related sections.

The *thermisch* sub-model is simulated in Matlab/Simulink and the data exchange with the Coordinator happens via SOAP. The *maschinen* sub-model is simulated in OpenModelica and exchanges data with the Coordinator via OPC UA. The following sections present the results obtained by simulating the proof-of-concept model in five different scenarios.

It should be noted that the times used for the macro-steps in the following scenarios are not exact times. For example, for the $100s$ macro-step, a value of 100.003 was chosen. This was due to the fact that certain events would not fire in the OpenModelica simulation if it was started at exactly the event time. For example, an event at time $t = 300s$

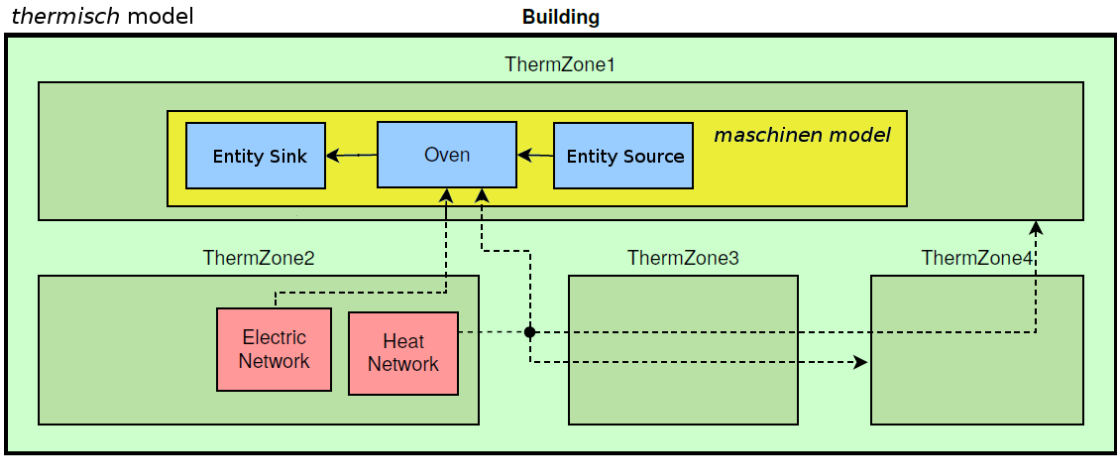


Figure 5.5: Overview of the model used in this thesis

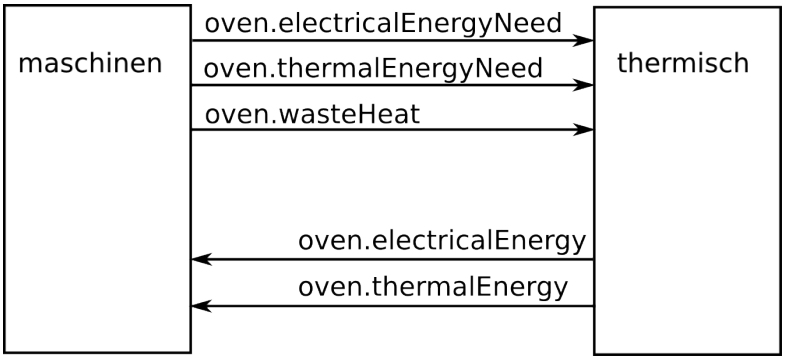


Figure 5.6: Simplified view of data exchange between the models

would not fire when the simulation was run with a start time of $300s$. This is possibly due to some implementation shortcomings in the event handling of the OpenModelica model. However, this does not diminish the proof-of-concept in any significant way, as the co-simulation strategies and framework still remain valid.

### 5.2.1 Scenario 1

This scenario tests the heating of the oven component. At $t = 0$, the oven wants to heat up, so it requests energy for heating right from the beginning. When the temperature of the **Oven** has reached the upper limit of the controller's hysteresis ($240°C$), it stops requesting thermal energy and the temperature begins to drop. When the temperature has reached the lower end of the hysteresis ($220°C$), the **Oven** starts requesting energy again. After $t = 25200s$ ($t = 7h$), the **Oven** is turned off and begins to cool down.

In the *weakly* coupled co-simulations (Figure 5.7), it can be seen that the **Oven** only starts warming up after two cycles, at times $t = 2 * macroStep$. This is due to the fact that before the first simulation step is run, the energy need is not yet communicated to the framework. So at the first data exchange with the *Coordinator* at $t = 0$, the OpenModelica framework, which manages the *maschinen* model, sends only the initial values of the variables, as defined in the `init.xml` file. Only after the first simulation step ($t = macroStep$) the OpenModelica framework sends the energy need to the *Coordinator*, which forwards it to the Matlab framework. After the exchange, the models are simulated again, and after this second cycle, the *thermisch* model outputs the energy requested by the *maschinen* model. In the 1000-second-interval co-simulation, the oven heats up to over 300 degrees because it takes one simulation cycle for the *thermisch* part to stop sending energy to the oven.

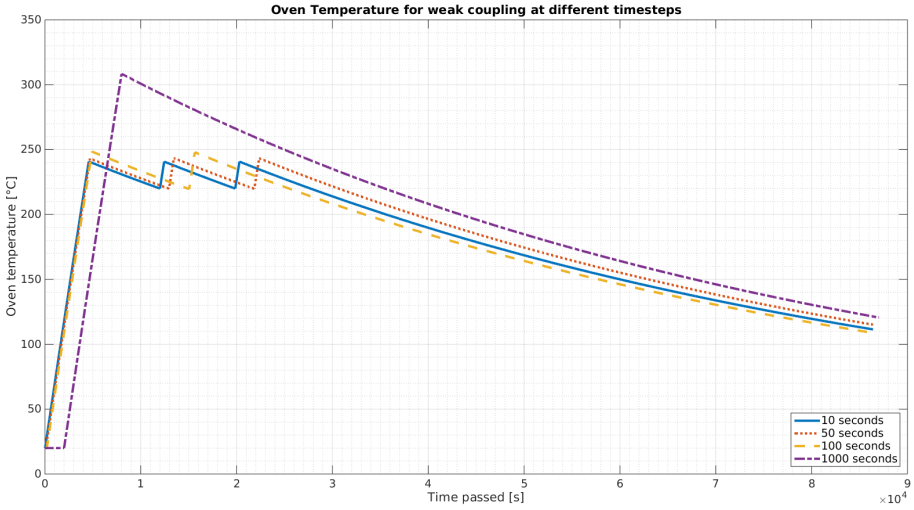


Figure 5.7: Oven temperature curves for weak coupling in scenario 1

Figure 5.8 shows a comparison of macro-steps and iteration count for *dynamic-parallel* coupling. Iteration 1 is a weakly *coupled* co-simulation with the appropriate macro-step size. For the $1000s$ macro-step, the temperature curve is pretty different than the original one. Even iterations 2 and 3 show an irreconcilable difference to the original. Interesting here is that iteration 2 produces the same result as *weak* coupling with a macro-step of $500s$. This behaviour is similar to the two-macro-step delay in the *weakly* coupled co-simulations mentioned above. After the first iteration, the **Oven** sends its thermal energy request to the *thermisch* simulation. The *thermisch* simulation received the request at the start of the second iteration and outputs the thermal energy at the end of the second iteration. The **Oven** thus receives its requested energy at the end of the second iteration, at which a new macro-step starts with iteration 1. So instead of a two-macro-step delay, there is only a one-macro-step delay but it still takes two iterations for the energy to reach the oven. In the third iteration, the models can already

interpolate the received thermal energy. The increase in **Oven** temperature begins when it is supposed to, at $t = 0s$, but the gradient starts slow and needs some time to reach the intended value. This behavior is due to the linear interpolation in the models. The received power is interpolated linearly inside the *maschinen* model, thus, the received power increases with time. This leads to an increase in computed temperature in each calculation step and a smoother temperature curve.

The $100s$ macro-step values in iteration 3 are getting pretty close to the desired results, but still don't quite match them. A major problem here is that there seems to be some problem with the temperature controller once the oven temperature reaches the lower threshold when cooling down.

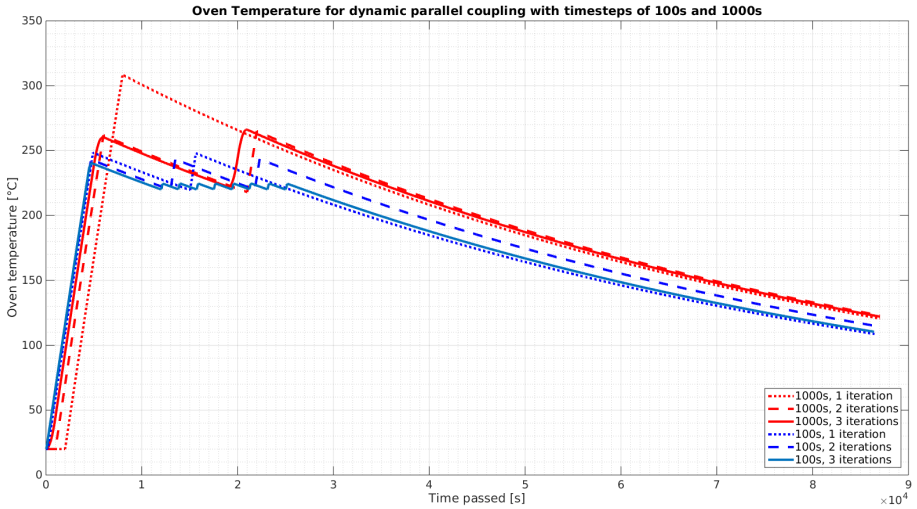


Figure 5.8: Comparison of iterations for dynamic parallel coupling in scenario 1

Figure 5.9 shows a comparison of macro-steps and iteration count for *dynamic-serial* coupling. In this mode, the first iteration is identical to the second iteration of *dynamic-parallel* coupling and the second iteration is identical to the third iteration of *parallel*. The *serial* co-simulation is essentially one iteration faster than the *parallel* co-simulation. This behaviour can be explained as follows:

First, the *parallel* co-simulation is examined:

1. The co-simulation starts with iteration 1 at $t = 0$, no energy is requested from the *thermisch* simulation, so at the end of the iteration, the *thermisch* simulation outputs 0 energy to the *maschinen* simulation.

2. The two models exchange data, the *maschinen* simulation receives 0 energy for the **Oven**, but the *thermisch* simulation is now aware of the energy needed by the **Oven**.

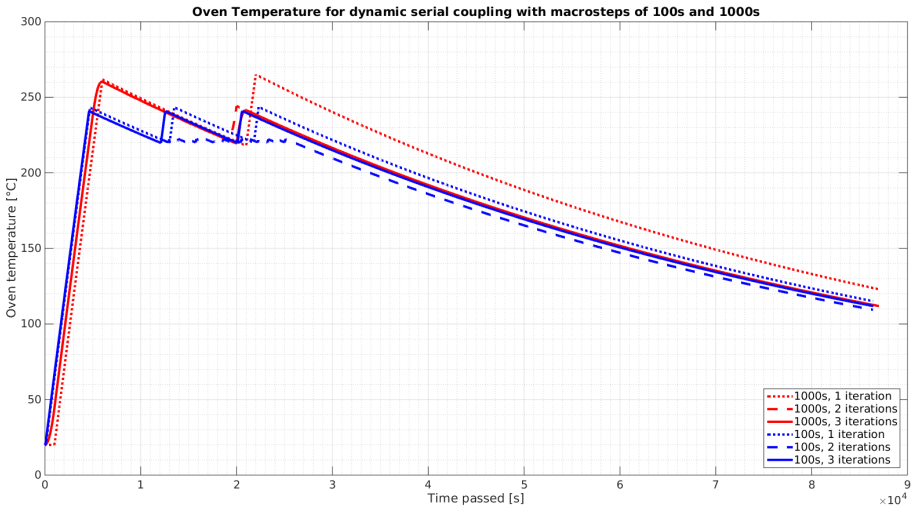


Figure 5.9: Comparison of iterations for dynamic serial coupling in scenario 1

3. Both models are simulated in iteration 2. The **Oven** still has not received any energy, so it does not heat up. At the end of this iteration, the *thermisch* simulation outputs the energy requested by the **Oven**.

4. The two models exchange data again, this time the *maschinen* simulation receives an interpolation vector for the requested energy which has a positive gradient.

5. Both models are simulated in iteration 3. The *maschinen* simulation can now interpolate a value for the received energy for the **Oven** based on the given interpolation vector. The temperature of the **Oven** begins to rise immediately.

This same behavior is seen one iteration earlier in the *serial* co-simulation if the *thermisch* model is simulated after the *maschinen* model:

1. The *maschinen* model is simulated first in iteration 1, the result is an energy need by the **Oven**.

2. The *thermisch* model is simulated in iteration 1 and already knows of the energy need by the **Oven**. The simulation outputs an interpolation vector with a positive gradient for the energy supplied to the **Oven**.

3. The *maschinen* model is simulated in iteration 2 and has already received the interpolation vector for the energy supplied to the **Oven** and can therefore already begin to interpolate the received energy.

An interesting phenomenon here is that the controller failure happens when each macro-step is iterated two times, but vanishes when three iterations are performed with the same macro-step.

Due to the fact that the simulations have to wait for one another in *serial* mode, there is no real simulation-time benefit to the reduced number of iterations. With 2 participating simulations and 2 iterations per macro-step, one macro-step takes 4 simulation cycles to complete. The number of simulation cycles grows by 1 for each participating simulation and iteration. At 2 iterations, one additional simulation in the co-simulation causes an increase in time of 2 simulation cycles. A *parallel* co-simulation with 2 participating simulations and 3 iterations only takes 3 simulation cycles to complete one macro-step. In theory, as Figure 5.10 shows, a *serial* co-simulation could be faster if some participating simulations are fast enough.

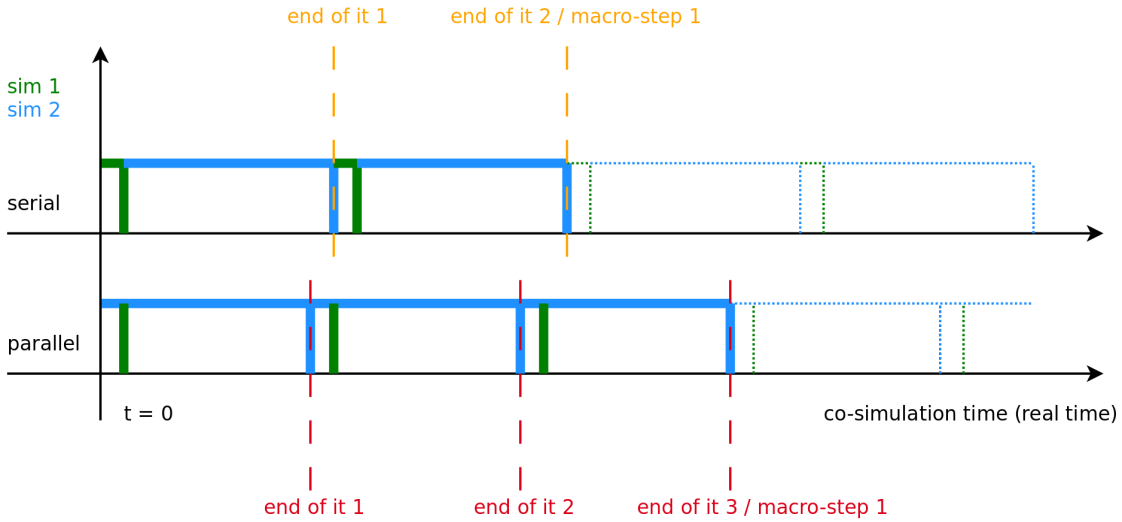


Figure 5.10: Serial and parallel co-simulation time for similar results

Figure 5.11 and Figure 5.12 show comparisons of the weakly and dynamically coupled results to the original results. The *dynamic-parallel* and the *dynamic-serial* co-simulations were simulated with a macro-step of 1000 seconds. In the 2-iteration results, the *dynamic-parallel* co-simulation is, again, basically just *weakly* coupled co-simulations with a macro-step of 500$s$. Due to this behaviour, the results of the parallely-coupled co-simulation are quite far off from the original.

The serially-coupled results seem to be a little bit better, but still deviate significantly from the original, which indicates that the macro-step size has to be lowered.

The weakly-coupled results start of pretty good, but the error accumulates pretty fast due to the flat decline in temperature when cooling down. By heating up just a bit more above the target temperature, the cooling takes much longer and the whole curve is streched out to the right.

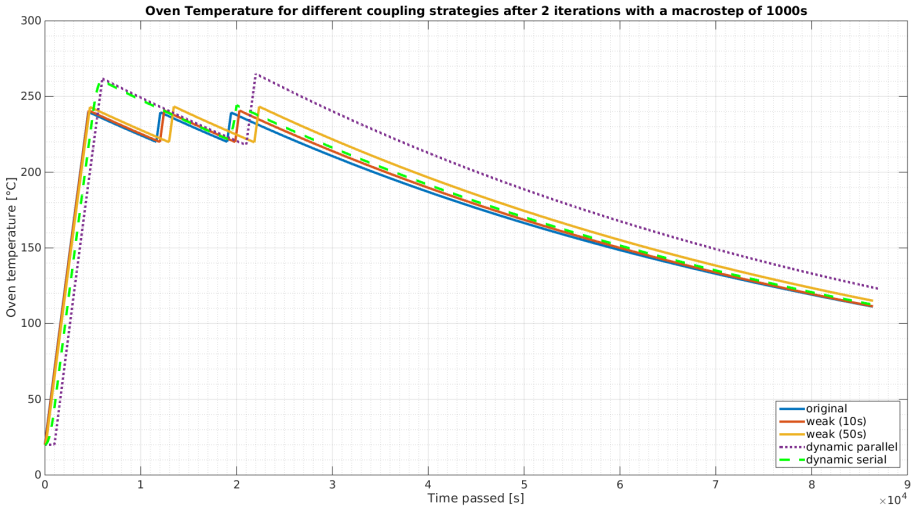


Figure 5.11: Comparison of different coupling strategies after 2 iterations for a macro-step of 1000 seconds in scenario 1

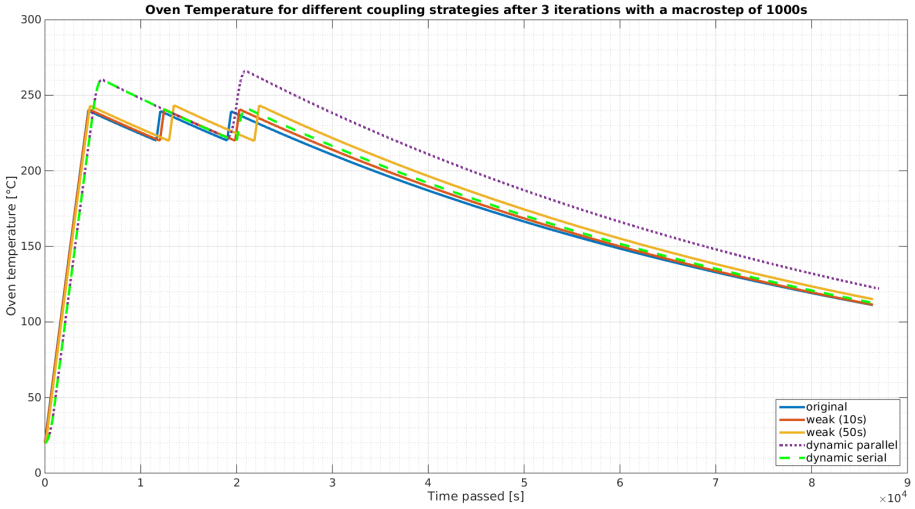


Figure 5.12: Comparison of different coupling strategies after 3 iterations for a macro-step of 1000 seconds in scenario 1

It can be seen in Figure 5.12 that the *parallel-dynamic* simulation has interpolated the thermal energy and the temperature curve is much smoother than after two iterations.

Figure 5.13 and Figure 5.14 show the results with a macro-step of 100 seconds. The controller failure can be seen again after two iterations in the *dynamic-serial* results, and after three iterations in the *dynamic-parallel* results.

Other than that, the results are much closer to the original data. Especially the *dynamic-serial* results after two/three iterations are very close to the *weak* coupling results with a macro-step of 10 seconds. This means that *dynamic-serial* coupling is producing similar results to *weak* coupling, but with a much higher macro-step size.

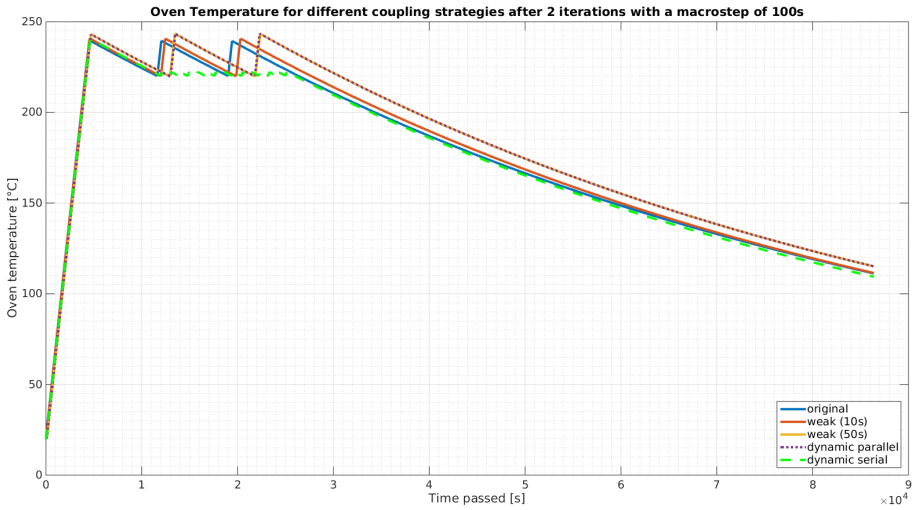


Figure 5.13: Comparison of different coupling strategies after 2 iterations for a macro-step of 100 seconds in scenario 1

To show that the *dynamic* coupling modes converge to the correct results by using smaller macro-step sizes, the scenario was simulated in *dynamic-serial* mode with 2 iterations and macro-steps of 5, 10 and 20 seconds. The scenario could not be simulated fully because of the time restriction on the trial version of the Unified Automation [Aut] OPC UA server, but Figure 5.15 and Figure 5.16 show the rise of the **Oven** temperature and the point where the temperature begins to drop again.

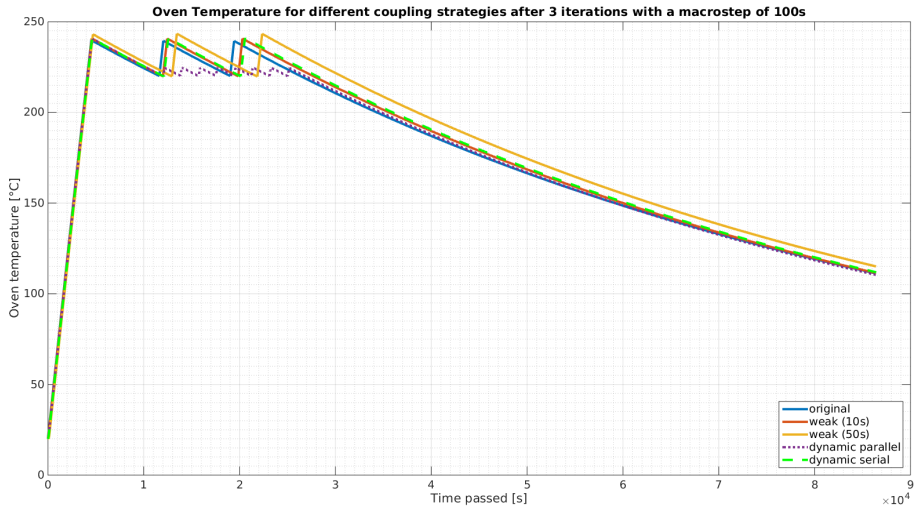


Figure 5.14: Comparison of different coupling strategies after 3 iterations for a macro-step of 100 seconds in scenario 1

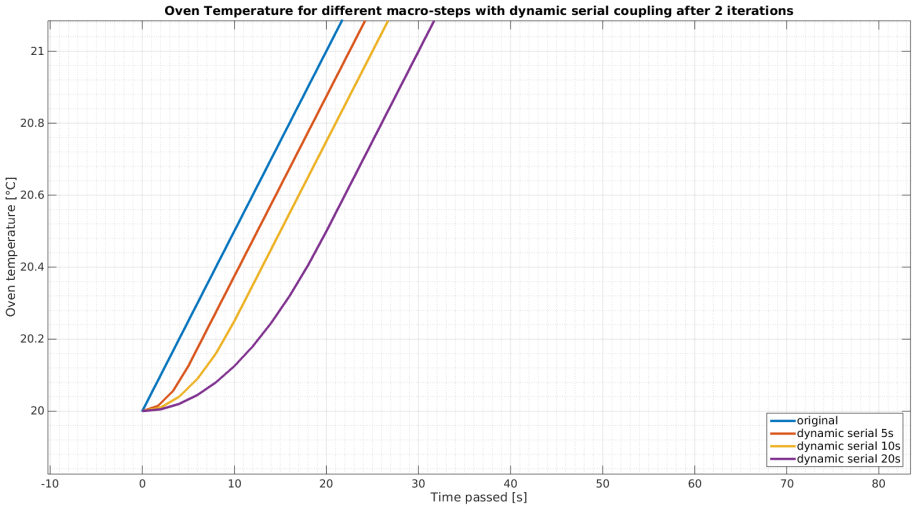


Figure 5.15: Temperature rise for different macro steps in dynamic-serial coupling mode in scenario 1

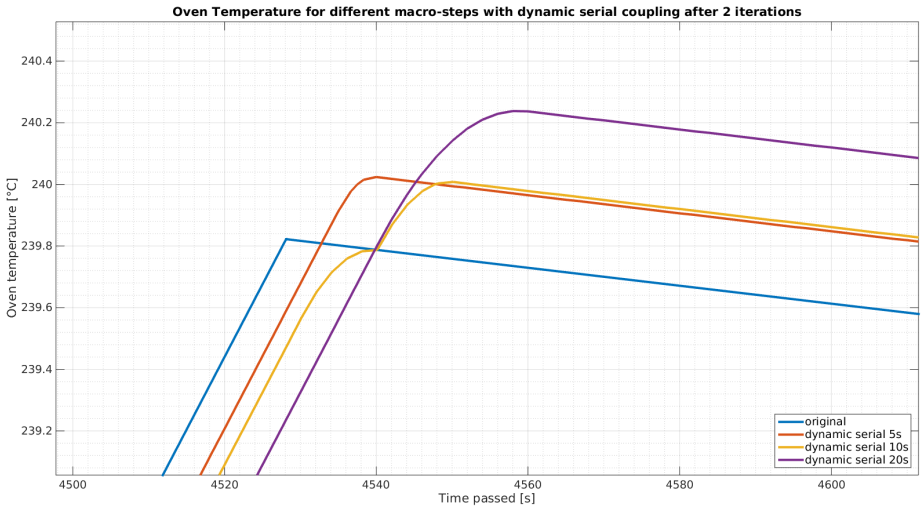


Figure 5.16: Temperature drop for different macro steps in dynamic-serial coupling mode in scenario 1

### 5.2.2 Scenario 2

In the second scenario, the **Oven** should start heating up at time $t = 0$ and be able to accept entities as soon as it has reached operating temperature ($230°C$), at about $t = 4320s$. The entity source outputs entities from time $t = 5400s$ until $t = 10800s$ with an interval of $180s$. The oven has a conveyor size of 5, this means that an entity remains $5 * 180s = 900$ seconds inside the **Oven** before reaching the output. Figure 5.17 shows the *weak* coupling results. Notice that with a $1000s$ macro-step, the **Oven** reaches operating temperature long after the entity source outputs entities into it. After accepting entities onto its conveyor belt, the **Oven** heats up and cools down much slower than before, due to the heat capacity of the entities inside it. Figure 5.18 shows the amount of entities processed. The input values increase each time the entity source offers an entity to the input of the **Oven**. The conveyor load of the **Oven** increases each time the **Oven** accepts an incoming entity and puts it on its conveyor. The output increases each time the entity sink receives an entity from the oven. At time $5400s + 900s = 6300s$, the first entity on the conveyor reaches the output of the oven and is output into the entity sink.

With a macro-step of $100s$, everything goes as expected, the **Oven** heats up fast enough to start accepting entities as soon as they arrive. As Figure 5.19 shows, problems start occurring with longer macro-step sizes. With *weak* coupling and a macro-step size of $1000s$, the oven does not reach operating temperature in time and has to reject incoming entities until the temperature is reached. Due to this delay, 6 fewer entities are processed. As opposed to *weak* coupling, *dynamic-parallel* and *dynamic-serial* coupling both output

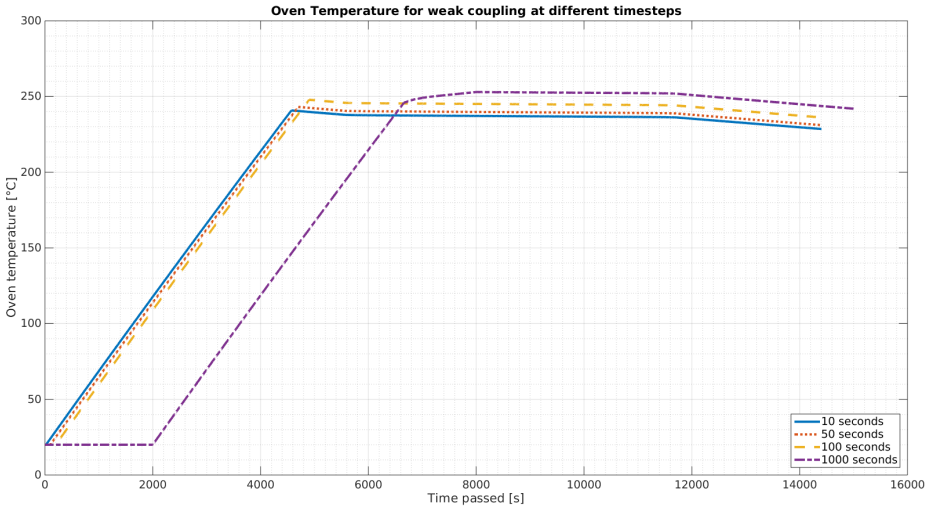


Figure 5.17: Oven temperature curves for weak coupling in scenario 2

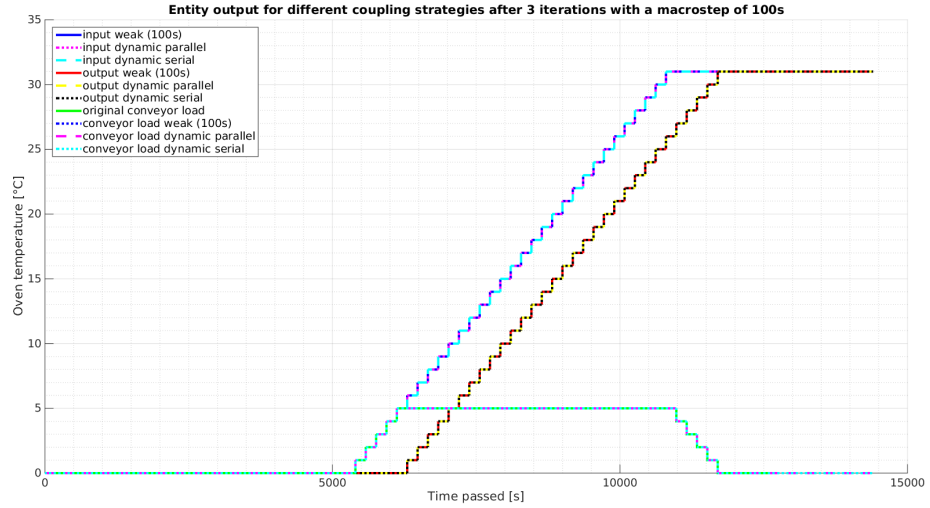


Figure 5.18: Comparison of entity throughput for a macro-step of 100s in scenario 2

the correct amount of entities because they heat up fast enough before entity source starts outputting entities.

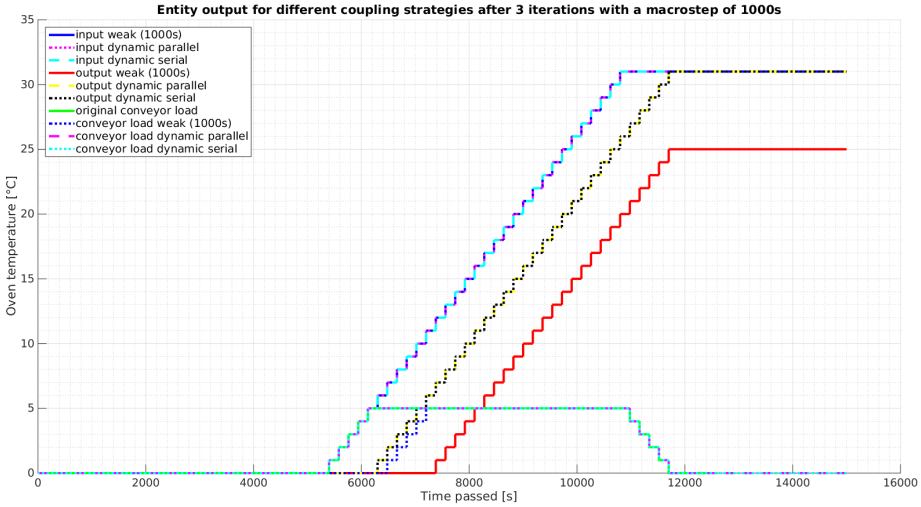


Figure 5.19: Comparison of entity throughput for a macro-step of 1000s in scenario 2

### 5.2.3 Scenario 3

Scenario 3 is very similar to scenario 2, the **Oven** starts heating up at $t = 0$, but it only processes a single entity at time $t = 4500s$. It stops heating at time $t = 6300s$ and slowly cools down until the simulation is stopped at time $t = 7200s$. Figure 5.20 shows that for a macro-step of $100s$, all coupling strategies produce results that match the original very closely, especially in the cooling phase, at $t > 5000s$.

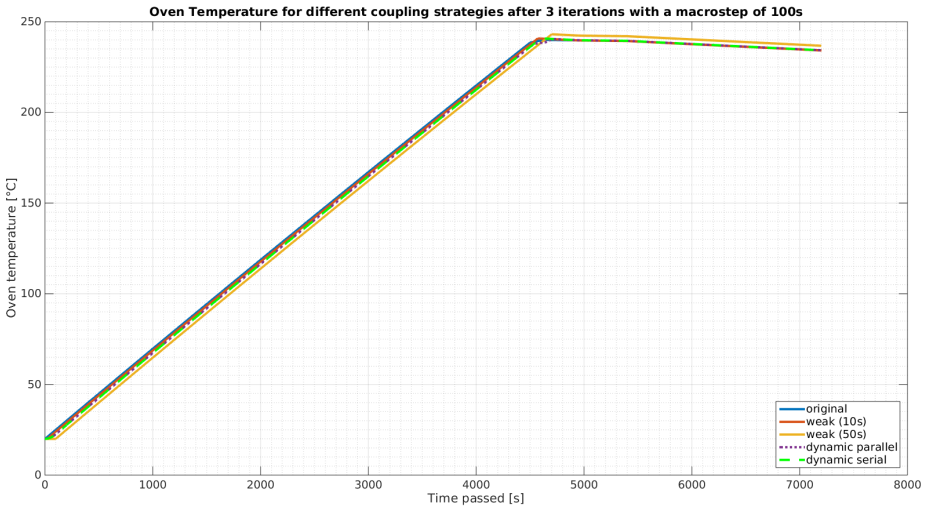


Figure 5.20: Comparison of different coupling strategies after 3 iterations for a macro-step of 100 seconds in scenario 3

### 5.2.4 Scenario 4

Scenario 4 is also pretty similar to the previous scenarios, with the difference being that entities are sent to the oven before it reaches its working temperature. In such a case, the oven rejects the entities until the working temperature is reached. If an entity is rejected, the entity source discards it and sends the next entity after the next interval has elapsed. This way, entities can only arrive at the oven at times that are a multiple of the interval, in this case $t = 180 * x$.

The oven starts heating at time $t = 0$, entities are sent from time $t = 3600s$ until $t = 7200s$. The oven should, again, reach working temperature at about $t = 4320s$ and start processing entities from this point onwards.

Because the entities are beeing sent at specific intervals, even small deviations in the time it takes to reach operating temperature can lead to differences in the amount of entities that are processed. Figure 5.21 shows the temperatures for a macro-step of 100

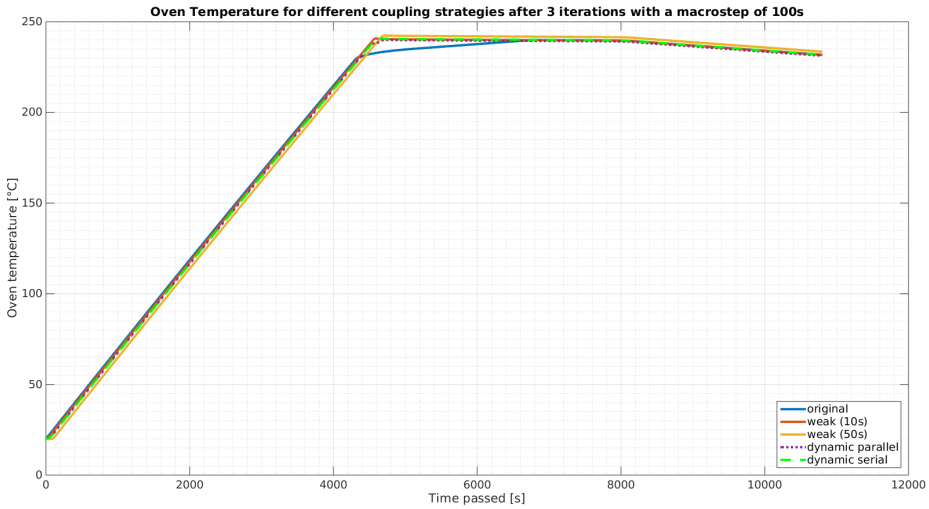


Figure 5.21: Comparison of different coupling strategies after 3 iterations for a macro-step of 100 seconds in scenario 4

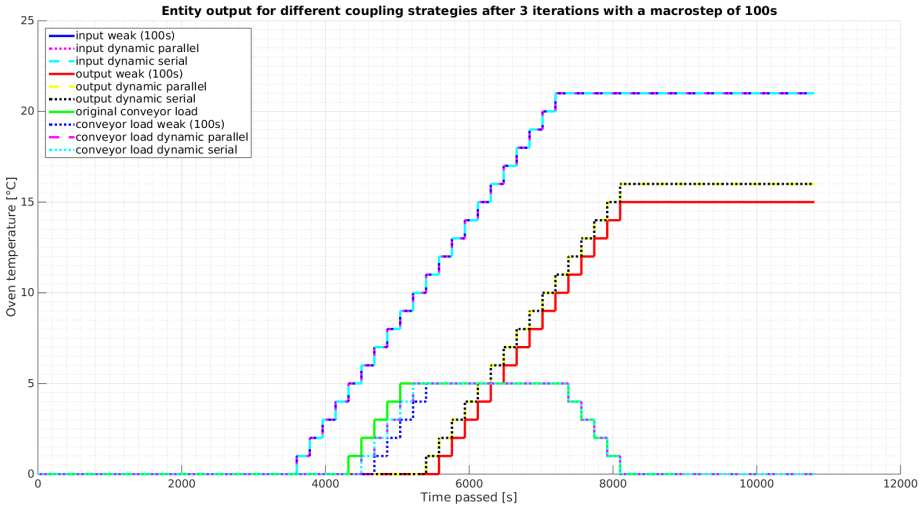


Figure 5.22: Comparison of entity throughput for a macro-step of 100s in scenario 4

seconds. Figure 5.22 shows the corresponding entity throughput. Due to the slightly faster temperature increase in the *dynamic-parallel* and *dynamic-serial* co-simulations, the **Oven** can process one more entity as opposed to *weak* coupling. Even so, the slight delay in the heating of the **Oven** causes less entities to be processed than in the original.

### 5.2.5 Scenario 5

In scenario 5, the **Oven** is used as a cooler, it greatly resembles scenario 1, but instead of positive thermal energy, the *maschinen* simulation requests negative thermal energy from the *thermisch* simulation in order for the temperature to drop inside the **Oven**. Figure 5.23 shows the results with a 100 seconds macro-step. Again, the temperature controller error can be observed in dynamic parallel mode after three iterations.

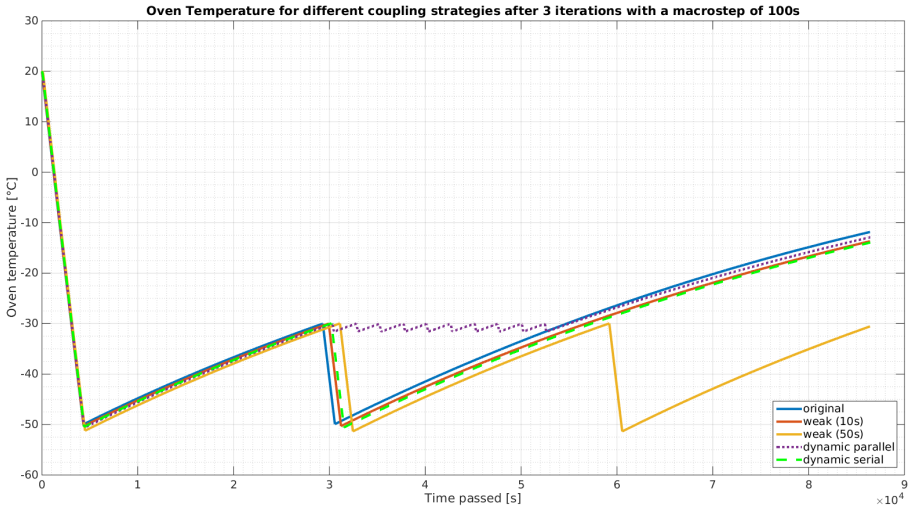


Figure 5.23: Comparison of different coupling strategies after 3 iterations for a macro-step of 100 seconds in scenario 5

## 5.3 Summary

To summarize, the results of the dynamic coupling strategies show much promise. Even with macro-step 10 times longer, the results are nearly as good as the weak coupling results. Additionally, dynamic-serial coupling can in certain scenarios be faster than dynamic-parallel coupling even if the simulations have to wait for one another.

Table 5.1 shows the performance results of the simulation tools and communication protocols. While it was obvious that SOAP would require more time for the data to be sent and processed, the actual time difference between the two protocols is immense. In real world applications, SOAP might often be unusable for these purposes.

Table 5.1: Performance comparison of SOAP and OPC UA

| | Matlab/SOAP | OM/OPC UA |
|---|---|---|
| Avg. time for single simulation | ~170ms | ~22ms |
| Avg. time for data transfer | ~115ms | ~2ms |
| approx. # of bytes for sending one datapoint | ~200 | ~690 |
| approx. # of bytes for receiving one datapoint | ~565 | ~360 |

CHAPTER 6

# Conclusion & Outlook

In conclusion, the current implementation works quite well and can, in theory, scale to very large numbers of simulations without much performance loss. This is because of the same timestep that the Coordinator uses for all simulations to control the co-simulation. If the co-simulation is executed in parallel, and all simulations run on individual hardware, then the only bottlenecks are the available bandwidth and the available memory/processing power of the Coordinator. As the processing of the co-simulation data inside the Coordinator can be done very quickly, it should, even with a large number of connected simulations, perform quite fast in relation to the time it takes for a simulation to execute. A downside of parallel co-simulation is, that a low number of simulations in the co-simulation setup take as much time as a high number of simulations to execute one simulation because each iteration takes as long as the slowest simulation.

In contrast to existing solutions which use inter-process and low-level socket communication, the communication technology used in this thesis (SOAP, OPC UA) transfers data at a much higher level and exchanging data is thus quite a bit slower. This offers other benefits, such as a greater extendability of the protocol. New datapoints can be included without modifying the existing protocol and new metadata and settings can be included with minimal changes in the source code. Also, restarting the simulations for each timestep and iteration is very time-consuming, but offers the possibility of dynamic-coupling.

The choices in this thesis are not so much about performance, but rather the new possibilities in terms of usability and extendability that these technologies and methods provide. This work shows that using high-level approaches can simplify the creation of a co-simulation. With the use of the developed frameworks, models barely have to be changed for them to be used in a co-simulation setup. Different parts of the model can be developed completely independently and need only specify the datapoints which they require from other parts. By using high level data transfer protocols and clearly defined interfaces between the simulations, the transferred data has clear semantics

which can be used to automate a number of things. An example of this would be to automatically convert engineering units between participating simulations. Additionally, by making the flow of the co-simulation dependent on the Coordinator, the coupling type for a co-simulation can be changed by simply adjusting the initialization file of the Coordinator.

These things greatly enchance usability and the ability to setup a co-simulation, but there is definitely room for improvement. For now, the data exchange between simulations and coordinator happens in fixed intervals defined in the configuration file of the coordinator. A first step to improve the design could be to implement adaptive timesteps, where the Coordinator can dynamically change the duration of a timestep based on some metric. Such a metric could, for example, be based on the rate of change of a specific variable. Or the Coordinator computes the rates of change of all variables and shortens/lengthens the timestep based on this. Such a behaviour could potentially lead to improvements in simulation time, as the timestep could be lengthened drastically in periods where no significant change occurs. Another bonus to this approach would be that data would/could be far more accurate in periods where the values of datapoints change fast. Although, the higher the number of connected simulations, i.e. the complexity of the co-simulation, the higher the chance that there will always be a variable which changes so that the timestep will be shortend, and the co-simulation will end up taking far longer than with a fixed timestep.

Another modification would be to implement a sort of distributed event detection, where the simulations exchange data before any event that uses a variable of another co-simulation. For example, if a simulation $A$ reaches an event $A.e$ at time 3.9 of timestep $3 - 4$, and a calculation is performed in the course of this event which uses a value from another co-simulation, then this value might have changed significantly in the other co-simulation, but the new value has not reached simulation $A$ yet. By exchanging data right before the event, the calculations of simulation $A$ will be far more accurate. Such an event-detection mechanism has been researched an implemented by Lin et. al. [Lin+11] for a smart-grid co-simulation.

One improvement, which might prove quite useful, would be to be able to send vectors of values as one datapoint. The current implementation only allows for one value per datapoint to be transmitted. Although the basics for this feature are already implemented in the WSDL file and the Coordinator, it has not yet been implemented in the individual frameworks.

To enhance extendability, it would be advantageous to implement a mechanism to choose a `SimulationRunner` (Matlab), `SimulationInterface` (OPC UA), `Simulation Connector` (Coordinator) and a `UnitConverter` (Coordinator) based on settings in the init XML files. With such a mechanism in place, a user would easily be able to choose which type of framework should run what type of simulation. Assuming that a new simulation tool is implemented for the OPC UA framework, the user could then choose in the init XML file if the OPC UA server should execute an OpenModelica or the other type of simulation.

# List of Figures

# Bibliography


[AG01] Martin Arnold and Michael Günther. "Preconditioned dynamic iteration for coupled differential-algebraic systems". In: *BIT Numerical Mathematics* 41.1 (2001), pp. 1–25.

[Aro] Jouni Aro. *OPC UA stack protocols*. URL: https://www.prosysopc.com/blog/opc-ua-1-02/ (visited on 11/26/2017).

[Ass] Modelica Association. *Modelica Homepage*. URL: https://www.modelica.org/ (visited on 11/26/2017).

[Aut] Unified Automation. *Unified Automation - Homepage*. URL: https://www.unified-automation.com/ (visited on 11/26/2017).

[Awa15] Muhammad Usman Awais. "Disctributed Hybrid Co-Simulation". In: (2015).

[BB14] S. Bandyopadhyay and R. Bhattacharya. *Discrete and Continuous Simulation: Theory and Practice*. Taylor & Francis, 2014. ISBN: 9781466596399.

[Blo+11] T. Blochwitz et al. "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". In: 2011.

[Boa10] IEEE-SA Standards Board. "IEEE Standard for Modeling and Simulation, High Level Architecture (HLA)– Framework and Rules". In: *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*. Aug. 2010, pp. 1–38.

[Cir+14a] Selim Ciraci et al. "FNCS: A Framework for Power System and Communication Networks Co-simulation". In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*. DEVS '14. Tampa, Florida: Society for Computer Simulation International, 2014, 36:1–36:8.

[Cir+14b] S. Ciraci et al. "Synchronization Algorithms for Co-simulation of Power Grid and Communication Networks". In: *Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*. Sept. 2014, pp. 355–364.

[CK06] BFrançois E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer US, 2006. ISBN: 9780387302607.

[CK13] Byoung Kyu Choi and Donghun Kang. *Modeling and Simulation of Discrete-Event Systems*. John Wiley & Sons, Inc., 2013. ISBN: 9781118732793. DOI: 10.1002/9781118732793. URL: http://onlinelibrary.wiley.com/book/10.1002/9781118732793.

[CMa] CMake. *CMake - Homepage*. URL: https://cmake.org/ (visited on 11/26/2017).

[Cona] Open Source Modelica Consortium. *OpenModelica Homepage*. URL: https://openmodelica.org/ (visited on 11/26/2017).

[Conb] World Wide Web Consortium. *World Wide Web Consortium Homepage*. URL: https://www.w3.org (visited on 11/26/2017).

[Don+09] Zhang Dong-xu et al. "Co-simulation with AMESim and MATLAB for differential dynamic coupling of Hybrid Electric Vehicle". In: *Intelligent Vehicles Symposium, 2009 IEEE*. June 2009, pp. 761–765.

[Ele13] General Electric. *Positive Sequence Load Flow - Homepage*. 2013. URL: http://www.geenergyconsulting.com/practice-area/software-products/pslf-re-envisioned (visited on 11/26/2017).

[Eng] Robert A. van Engelen. *gSOAP - Homepage*. URL: https://www.cs.fsu.edu/~engelen/soap.html (visited on 11/26/2017).

[Foua] Apache Soaftware Foundation. *Apache CXF Homepage*. URL: http://cxf.apache.org/ (visited on 11/26/2017).

[Foub] Free Software Foundation. *GNU Project Homepage*. URL: https://www.gnu.org/home.en.html (visited on 11/26/2017).

[Fouc] OPC Foundation. *OPC Foundation Homepage*. URL: https://opcfoundation.org/ (visited on 11/26/2017).

[Groa] W3C XML Protocol Working Group. *SOAP Message Transmission Optimization Mechanism*. URL: https://www.w3.org/TR/soap12-mtom/ (visited on 11/26/2017).

[Grob] W3C XML Protocol Working Group. *XML Optimized Packaging*. URL: https://www.w3.org/TR/2005/REC-xop10-20050125/ (visited on 11/26/2017).

[HHR13] Irene Hafner, Bernhard Heinzl, and Matthias Rössler. "An Investigation on Loose Coupling Co-Simulation with the BCVTB". In: *Simulation Notes Europe SNE* 23 (2013), pp. 45–50.

[Kel+15] B.M. Kelley et al. "A federated simulation toolkit for electric power grid and communication network co-simulation". In: *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2015 Workshop on*. Apr. 2015, pp. 1–6.

[LA11] V. Liberatore and A. Al-Hammouri. "Smart grid communication and co-simulation". In: *Energytech, 2011 IEEE*. May 2011, pp. 1–5.

[Lin+11] Hua Lin et al. “Power system and communication network co-simulation for smart grid applications”. In: *Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES*. Jan. 2011, pp. 1–6.

[Mat] MathWorks. *MathWorks Homepage*. URL: https://mathworks.com/ (visited on 11/26/2017).

[Nou] Thierry S. Nouidui. *BCVTB Homepage*. URL: https://simulationresearch.lbl.gov/bcvtb (visited on 11/26/2017).

[Püh16] Clemens Pühringer. *Using the Modelica language to simulate hybrid models*. Tech. rep. Student project. TU Wien, Oct. 2016.

[Rai+16] Philipp Raich et al. “Modeling Techniques for Integrated Simulation of Industrial Systems Based on Hybrid PDEVS”. In: *2016 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. Apr. 2016, pp. 1–6. DOI: 10.1109/MSCPES.2016.7480221.

[Sch15] Robert Schmoll. “Co-Simulation und Solverkopplung, Analyse komplexer multiphysikalischer Systeme”. In: (2015).

[Smo+16] P. Smolek et al. “Hybrid Building Performance Simulation Models for Industrial Energy Efficiency Applications”. In: *Proceedings of the 11th Conference on Sustainable Development of Energy, Water and Environment Systems (SDEWES 2016)*. 2016.

[Sou] University of Southern California - Information Sciences Institute. *Network Simulator 2 - Homepage*. URL: http://www.isi.edu/nsnam/ns/ (visited on 11/26/2017).

[tea] The Wireshark team. *Wireshark Homepage*. URL: https://www.wireshark.org/ (visited on 11/26/2017).

[Tho] Lee Thomason. *TinyXML2 Github page*. URL: https://github.com/leethomason/tinyxml2 (visited on 11/26/2017).

[Ton10] Xiaoyang Tong. “The Co-simulation Extending for Wide-area Communication Networks in Power System”. In: *Power and Energy Engineering Conference (APPEEC), 2010 Asia-Pacific*. Mar. 2010, pp. 1–4.

[Wei08] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design (The MK/OMG Press)*. 1st ed. Morgan Kaufmann, Feb. 2008. ISBN: 9780123742742.

[Wet12] Michael Wetter. “Co-Simulation of Building Energy and Control Systems with the Building Controls Virtual Test Bed”. In: *Journal of Building Performance Simulation*. Sept. 2012, pp. 185–203.

[Whi+85] J. White et al. “Waveform Relaxation: Theory and Practice”. In: *Transactions of the Society for Computer Simulation* 2.1 (1985), pp. 95–133.

# Appendix A: Example Initialization Files

## Coordinator Initialization File

```
<?xml version="1.0"?>
<cosimulation>
  <coupling>
    <mode>weak</mode>
    <iterations>2</iterations>
    <precision>0.001</precision>
  </coupling>
  <timing>
    <startTime>0</startTime>
    <endTime>86400</endTime>
    <timeStep>50.00001</timeStep>
    <maxVariation>5.0</maxVariation>
  </timing>
  <solver>
    <name>ode45</name>
    <absoluteTolerance>0.000000001</absoluteTolerance>
    <relativeTolerance>0.00000001</relativeTolerance>
    <minimumStepSize>1</minimumStepSize>
    <maximumStepSize>5</maximumStepSize>
  </solver>
  <simulations>
    <simulation>
      <name>thermisch</name>
      <ordering>2</ordering>
      <connection-type>SOAP</connection-type>
      <solver>
        <solver>ode45</solver>
        <absoluteTolerance>0.05</absoluteTolerance>
        <minimumStepSize>5</minimumStepSize>
        <maximumStepSize>10</maximumStepSize>
      </solver>
    </simulation>
```

```
    <simulation>
      <name>maschinen</name>
      <ordering>1</ordering>
      <connection-type>OPCUA</connection-type>
      <connection-information>opc.tcp://localhost:48010</connection-information>
      <solver>
        <solver>dassl</solver>
        <absoluteTolerance>0.04</absoluteTolerance>
        <minimumStepSize>2</minimumStepSize>
      </solver>
    </simulation>
  </simulations>
</cosimulation>
```

## OpenModelica Framework Initialization File

```
<?xml version="1.0"?>
<cosimulation>
  <configuration>
    <solver>
      <solver>dassl</solver>
      <minimumStepSize>1</minimumStepSize>
    </solver>
  </configuration>
  <output>
    <simulation>
      <model-src>Cube/package.mo</model-src>
      <model-name>Cube.Production.MachinesScenario5</model-name>
      <name>maschinen</name>
      <datapoints>
        <datapoint>
          <name>TZ1_Q_AW_WM</name>
          <value>0</value>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>TZ1_Q_AW_O</name>
          <value>0</value>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>TZ1_PelB_O</name>
```

```
          <value>0</value>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>TZ1_Q_WB_O</name>
          <value>0</value>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>globalState</name>
          <value>0</value>
          <unit>
            <name>Unit</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>T</name>
          <value>0</value>
          <unit>
            <name>C</name>
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>Q_W</name>
          <value>0</value>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
      </datapoints>
    </simulation>
  </output>
  <input>
    <simulation>
      <name>thermisch</name>
      <datapoints>
        <datapoint>
          <name>TZ1_Pel_O</name>
          <unit>
            <name>W</name>
```

```
            <power>0</power>
          </unit>
        </datapoint>
        <datapoint>
          <name>TZ1_Q_W</name>
          <unit>
            <name>W</name>
            <power>0</power>
          </unit>
        </datapoint>
      </datapoints>
    </simulation>
  </input>
</cosimulation>
```

## Matlab Framework Initialization File

```
<?xml version="1.0"?>
<cosimulation>
  <output>
    <simulation>
      <name>thermisch</name>
      <dataset>
        <time>0</time>
        <datapoints>
          <datapoint>
            <name>TZ1_Pel_O</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>TZ1_Q_W</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>qwges</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>ttz1</name>
```

```
            <unit>
              <name>°C</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>ttz2</name>
            <unit>
              <name>°C</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>ttz3</name>
            <unit>
              <name>°C</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>ttz4</name>
            <unit>
              <name>°C</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>hbtz1</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>kbtz1</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
        </datapoints>
      </dataset>
    </simulation>
  </output>
  <input>
    <simulation>
      <name>maschinen</name>
      <dataset>
        <time>0</time>
```

```
        <datapoints>
          <datapoint>
            <name>TZ1_Q_AW_WM</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>TZ1_Q_AW_O</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>TZ1_PelB_O</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>TZ1_Q_WB_O</name>
            <unit>
              <name>W</name>
              <power>0</power>
            </unit>
          </datapoint>
          <datapoint>
            <name>T</name>
            <unit>
              <name>C</name>
              <power>0</power>
            </unit>
          </datapoint>
        </datapoints>
      </dataset>
    </simulation>
  </input>
</cosimulation>
```